

GI-Edition

Lecture Notes in Informatics

Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, Wolfram Wingerath (Hrsg.)

Datenbanksysteme für Business, Technologie und Web (BTW 2015)

04. – 06. März 2015
Hamburg

Proceedings

241

Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler,
Theo Härder, Steffen Friedrich, Wolfram Wingerath (Hrsg.): BTW 2015



Thomas Seidl, Norbert Ritter, Harald Schöning,
Kai-Uwe Sattler, Theo Härder, Steffen Friedrich,
Wolfram Wingerath (Hrsg.)

Datenbanksysteme für Business, Technologie und Web (BTW 2015)

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

04. – 06.03.2015
in Hamburg, Germany

Gesellschaft für Informatik e.V. (GI)

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

Volume P-241

ISBN 978-3-88579-635-0
ISSN 1617-5468

Volume Editors
Thomas Seidl
Lehrstuhl Informatik 9
RWTH Aachen
52056 Aachen, Germany
E-Mail: seidl@informatik.rwth-aachen.de
Norbert Ritter
Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme
22527 Hamburg, Germany
E-Mail: ritter@informatik.uni-hamburg.de
Harald Schöning
Software AG
Uhlandstr. 12
64297 Darmstadt, Germany
E-Mail: harald.schoening@softwareag.com
Kai-Uwe Sattler
Technische Universität Ilmenau
FG Datenbanken und Informationssysteme Postfach 100 565
98684 Ilmenau, Germany
E-Mail: kus@tu-ilmenau.de
Theo Härder
Fachbereich Informatik
TU Kaiserslautern
67653 Kaiserslautern, Germany
E-Mail: haerder@informatik.uni-kl.de
Steffen Friedrich
Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme
22527 Hamburg, Germany
E-Mail: friedrich@informatik.uni-hamburg.de
Wolfram Wingerath
Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme
22527 Hamburg, Germany
E-Mail: wingerath@informatik.uni-hamburg.de

Series Editorial Board
Heinrich C. Mayr, Alpen-Adria-Universität Klagenfurt, Austria
(Chairman, mayr@ifit.uni-klu.ac.at)
Dieter Fellner, Technische Universität Darmstadt, Germany
Ulrich Flegel, Hochschule für Technik, Stuttgart, Germany
Ulrich Frank, Universität Duisburg-Essen, Germany
Johann-Christoph Freytag, Humboldt-Universität zu Berlin, Germany
Michael Goedicke, Universität Duisburg-Essen, Germany
Ralf Hofestädt, Universität Bielefeld, Germany
Michael Koch, Universität der Bundeswehr München, Germany
Axel Lehmann, Universität der Bundeswehr München, Germany
Peter Sanders, Karlsruher Institut für Technologie (KIT), Germany
Sigrid Schubert, Universität Siegen, Germany
Ingo Timm, Universität Trier, Germany
Karin Vosseberg, Hochschule Bremerhaven, Germany
Maria Wimmer, Universität Koblenz-Landau, Germany

Dissertations
Steffen Hölldobler, Technische Universität Dresden, Germany
Seminars
Reinhard Wilhelm, Universität des Saarlandes, Germany
Thematics
Andreas Oberweis, Karlsruher Institut für Technologie (KIT), Germany

Gesellschaft für Informatik, Bonn 2015
printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Die 16. Fachtagung „Datenbanksysteme für Business, Technologie und Web“ (BTW) des Fachbereichs „Datenbanken und Informationssysteme“ (DBIS) der Gesellschaft für Informatik (GI) findet vom 4. bis 6. März 2015 an der Universität Hamburg statt. Auf der BTW trifft sich im zweijährigen Rhythmus die deutschsprachige Datenbankgemeinde, um neue Fragestellungen zu erörtern und aktuelle Forschungsergebnisse zu präsentieren und zu diskutieren. Zu ihrem 30-jährigen Jubiläum ist die BTW zu Gast in Hamburg, wo schon 32 Jahre vorher im Jahr 1983 ein GI-Fachgespräch zum Thema „Sprachen für Datenbanken“ durchgeführt wurde, das sich als unmittelbarer Vorläufer der BTW-Konferenzreihe verstehen lässt.

Die jahrhundertelange Geschichte der Hansestadt Hamburg als Handelsmetropole bildet einen fruchtbaren Boden für den modernen, norddeutschen IT-Standort. Die enge Zusammenarbeit von Hochschulen, Instituten und Unternehmen ermöglicht und stärkt die Entwicklung von neuen Technologien zur Unterstützung sowohl von Handel und Dienstleistungen, aber auch vieler anderer Anwendungen in Natur-, Lebens-, Ingenieurs-, Medien- und Wirtschaftswissenschaften. Für all diese Bereiche bietet die aktuelle Datenbankforschung neue Lösungsansätze und die BTW deckt die volle Bandbreite von Modellierung, Management und Speicherung von Daten über Anfragebearbeitung, -optimierung und Benchmarking bis hin zu Zugriffsschutz und vielfältigen Analyseaufgaben ab. Dabei sind alle Systemebenen von der physischen Speicherung über Anfragesprachen bis zur semantischen Ebene vertreten. Klassische Themen stehen neben neuen Optionen, das Spektrum reicht von Hauptspeicherdatenbanken bis zu verteilten Cloud-Systemen über relationale Daten hinaus zu verlinkten und multimedialen Daten, von statischen Situationen zu Daten- und Anfrageströmen.

Für das wissenschaftliche Programm wurden aus 48 Einreichungen 25 für den vorliegenden Tagungsband und zur Präsentation auf der Tagung ausgewählt, davon 18 Lang- und 7 Kurzbeiträge in der oben ausgeführten thematischen Breite, die zu acht Sitzungen gruppiert werden. Erstmals auf der BTW wurde eine zweite Begutachtungsrunde durchgeführt, in der die Gutachter die Umsetzung der von ihnen empfohlenen Änderungen überprüften.

Im Industrieprogramm spiegelt sich die Aktualität des Themas „Prozesse“ auch unter dem Aspekt der Datenanalyse wider. Von vier Lang- und vier Kurzbeiträgen befasst sich mehr als die Hälfte damit oder zumindest generell mit dem Thema „Analytics“. Erfreulich ist, dass sich auch der akademische Bereich wieder den kommerziell verfügbaren und genutzten Systemen zuwendet und das verfügbare Spektrum analysiert. In einer Panel-Veranstaltung haben die Teilnehmer der Konferenz die Möglichkeit, die Antworten der „Big Player“ auf die

Frage „Big Data – Evolution oder Revolution in der Datenverarbeitung?" selbst zu vergleichen.

Für das Demoprogramm wurden 17 Beiträge eingereicht, von denen das Programmkomitee elf zur Live-Demonstration während der Konferenz auswählen konnte. Die Bandbreite der Demo-Themen reicht dabei von der Unterstützung der Datenbankausbildung über Kernthemen wie Indexing und Cluster-Betrieb bis hin zur Unterstützung der Entwicklung von Datenbankanwendungen, Data Mining und Recommender-Systemen.

Inhaltliche Höhepunkte bilden die eingeladenen Vorträge von IBM Fellow C. Mohan, von Alexander Böhm, Datenbankarchitekt SAP Hana, von Eric Sedlar, Vice President & Technical Director Oracle Labs, und von Jens Dittrich, Universität Saarbrücken, die inhaltlich den Bogen von Small Data bis Big Data spannen und aktuelle Herausforderungen sowohl aus Sicht der angewandten als auch der universitären Forschung benennen.

Eine Vorstellung mit Diskussion der beiden vom Bundesministerium für Bildung und Forschung (BMBF) kürzlich eingerichteten Kompetenzzentren BBDC (Berlin Big Data Center, Leitung TU Berlin) und ScaDS (Competence Center for Scalable Data Services and Solutions, Leitung der TU Dresden und Uni Leipzig) sowie des in Zusammenarbeit von Industrie und Forschung eingerichteten SDIL (Smart Data Innovation Lab, Leitung KIT) rundet das wissenschaftliche Programm ab.

Der BTW-Dissertationspreis des GI-Fachbereichs DBIS wird dieses Jahr zum achten Mal vergeben. Aus den mit der Prüfung im Zeitraum von Oktober 2012 bis September 2014 abgeschlossenen Promotionen wurden die Dissertationen „Forecasting in Database Systems“ von Frau Ulrike Fischer (Technische Universität Dresden, Betreuung Wolfgang Lehner) und „Efficient Querying and Learning in Probabilistic and Temporal Databases“ von Herrn Maximilian Dylla (Max-Planck-Institut für Informatik und Universität des Saarlandes, Betreuung Gerhard Weikum und Martin Theobald) für den Preis ausgewählt.

Im Vorfeld des Hauptprogramms finden Tutorien, Workshops und ein Studierendenprogramm statt. Die Informationen und Materialien zur BTW 2015 werden auch über die Tagung hinaus unter http://btw-2015.de zur Verfügung stehen.

Die Organisation der BTW als große Tagung mit vielen angeschlossenen Veranstaltungen ist nicht ohne die Unterstützung vieler Partner möglich. Diese sind auf den folgenden Seiten aufgeführt; zu ihnen zählen insbesondere die Sponsoren und auch die Geschäftsstelle der Gesellschaft für Informatik. Ihnen allen gilt unser besonderer Dank!

Aachen, Darmstadt, Ilmenau, Kaiserslautern, Hamburg im Januar 2015

Thomas Seidl, Vorsitzender des wissenschaftlichen Programmkomitees

Norbert Ritter, Tagungsleitung und Vorsitzender des Organisationskomitees

Harald Schöning, Vorsitzender des Industrieprogrammkomitees

Kai-Uwe Sattler, Vorsitzender des Demoprogrammkomitees

Theo Härder, Vorsitzender des Dissertationspreiskomitees

Wolfram Wingerath, Steffen Friedrich, Tagungsband und Organisationskomitee

Tagungsleitung

Norbert Ritter, Universität Hamburg

Organisationskomitee

Norbert Ritter
Anne Hansen-Awizen
Wolfram Wingerath
Steffen Friedrich
Felix Gessert
Fabian Panse
Volker Nötzold

Wissenschaftliches Programm

Vorsitz: Thomas Seidl, RWTH Aachen

Ira Assent, Univ. Aarhus
Wolf-Tilo Balke, TU Braunschweig
Christian Böhm, LMU München
Erik Buchmann, KIT Karlsruhe
Stefan Conrad, Univ. Düsseldorf
Stefan Dessloch, TU Kaiserslautern
Jens Dittrich, Univ. Saarland
Rainer Gemulla, Univ. Mannheim
Michael Gertz, Univ. Heidelberg
Theo Härder, TU Kaiserslautern
Alfons Kemper, TU München
Meike Klettke, Univ. Rostock
Harald Kosch, Univ. Passau
Georg Lausen, Univ. Freiburg
Ulf Leser, HU Berlin
Jens Teubner, TU Dortmund
Sebastian Michel, TU Kaiserslautern
Emmanuel Müller, KIT Karlsruhe
Felix Naumann, HPI Potsdam
Thomas Neumann, TU München
Daniela Nicklas, Univ. Bamberg
Erhard Rahm, Univ. Leipzig
Stefanie Rinderle-Ma, Univ. Ulm
Norbert Ritter, Univ. Hamburg
Eike Schallehn, Univ. Magdeburg
Ingo Schmitt, BTU Cottbus
Holger Schwarz, Univ. Stuttgart
Bernhard Seeger, Univ. Marburg
Uta Störl, Hochschule Darmstadt
Klaus Meyer-Wegener, Univ. Erlangen-Nürnberg

Dissertationspreise

Vorsitz: Theo Härder, TU Kaiserslautern

Johann-Christoph Freytag, HU Berlin
Andreas Heuer, Univ. Rostock
Alfons Kemper, TU München
Wolfgang Lehner, TU Dresden
Bernhard Mitschang, Univ. Stuttgart
Erhard Rahm, Univ. Leipzig
Gerhard Weikum, MPI Saarbrücken

Industrieprogramm

Vorsitz: Harald Schöning, Software AG

Aaron Auld, EXASOL
Henrik Loeser, IBM
Norman May, SAP
Thomas Ruf, freier IT Consultant
Knut Stolze, IBM

Demoprogramm

Vorsitz: Kai-Uwe Sattler, TU Ilmenau

Carsten Binnig, DHBW Mannheim
Peter Fischer, Univ. Freiburg
Torsten Grust, Univ. Tübingen
Dirk Habich, TU Dresden
Melanie Herschel, Univ. Paris-Sud
Katja Hose, Univ. Aalborg
Birgitta König-Ries, Univ. Jena
Holger Meye, Univ. Rostock
Knut Stolze, IBM Böblingen

Externe Gutachter

Can Altinigneli, Univ. München
Anas Alzoghbi, Univ. Freiburg
Victor Anthony Arrascue Ayala, Univ. Freiburg
Christian Authmann, Univ. Marburg
Christian Beilschmidt, Univ. Marburg
Kenneth S. Bøgh, Univ. Aarhus
Kristof Böhmer, Univ. Wien
Thomas Böttcher, TU Cottbus
Daniel Braun, Univ. Düsseldorf
Sebastian Breß, TU Dortmund
Johannes Drönner, Univ. Marburg
Walid Fdhila, Univ. Wien
Sebastian Goebl, Univ. München
Bastian Hoßbach, Univ. Marburg
Conrad Indiono, Univ. Wien
Martin Junghans, Univ. Leipzig
Georg Kaes, Univ. Wien
Matthias Liebeck, Univ. Düsseldorf
Michael Lind Mortensen, Univ. Aarhus
Andre Petermann, Univ. Leipzig
Niko Pollner, Univ. Erlangen
Martin Przyjaciel-Zablocki, Univ. Freiburg
Stefan Richter, Univ. Saarland
Magdalena Rischka, Univ. Düsseldorf
Sascha Saretz, TU Cottbus
Martin Schäler, Univ. Magdeburg
Alexander Schätzle, Univ. Freiburg
Stefan Schuh, Univ. Saarland
Felix Martin Schuhknecht, Univ. Saarland
Michael Singhof, Univ. Düsseldorf
Peter Wackersreuther, Univ. München
Holger Wegert, H Darmstadt
Marcel Zierenberg, TU Cottbus

Inhaltsverzeichnis

Eingeladene Vorträge

C. Mohan
Big Data: Hype and Reality 21

Alexander Böhm
Novel Optimization Techniques for Modern Database Environments 23

Eric Sedlar
When can "big data" be "in-memory"? 25

Jens Dittrich
The Case for Small Data Management 27

Wissenschaftliches Programm

Cloud Data Management

Andreas Göbel, Marcel Sufryd
KitMig – Flexible Live-Migration in mandantenfähigen Datenbanksystemen 33

Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Steffen Friedrich, Norbert Ritter
The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management 53

Michael Schaarschmidt, Felix Gessert, Norbert Ritter
Towards Automated Polyglot Persistence 73

Rights and Privacy

Ziad Sehili, Lars Kolb, Christian Borgs, Rainer Schnell, Erhard Rahm
Privacy Preserving Record Linkage with PPJoin 85

Jan Hendrik Nielsen, Daniel Janusz, Jochen Taeschner, Johann-Christoph Freytag
D2Pt: Privacy-Aware Multiparty Data Publication 105

Timo Michelsen, Marco Grawunder, Cornelius Ludmann, H.-Jürgen Appelrath
Admission Control für kontinuierliche Anfragen 125

Schema Engineering

Michael Schäfers, Udo W. Lipeck
Meeting the Challenges of Integrating Large and Diverse Geographic Databases 137

Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt, Wolfgang Lehner
Relationships for Dynamic Data Types in RSQL 157

Patrick Arnold, Erhard Rahm
SemRep: A Repository for Semantic Mapping 177

Physical Layer

Daniel Schall, Theo Härder
Energy and Performance - Can a Wimpy-Node Cluster Challenge a Brawny Server 197

Caetano Sauer, Goetz Graefe, Theo Härder
Single-pass restore after a media failure 217

Goetz Graefe, Hideaki Kimura
Orthogonal key-value locking 237

Query Processing

Dirk Habich, Patrick Damme, Wolfgang Lehner
Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse 259

Stefan Hagedorn, Katja Hose, Kai-Uwe Sattler
SPARQling Pig – Processing Linked Data with Pig Latin 279

Thomas Stadelmann, Daniel Blank, Andreas Henrich
Implementierung von IR-Modellen auf Basis spaltenorientierter Datenbanken oder invertierter Listen 299

Benchmarking

Moritz Kaufmann, Tobias Mühlbauer, Manuel Then, Andrey Gubichev, Alfons Kemper, Thomas Neumann
Hochperformante Analyse von Graph-Datenbanken 311

Cesare Pautasso, Dieter Roller, Frank Leymann, Vincenzo Ferme, Marigianna Skouradaki
Towards Workflow Benchmarking: Open Research Challenges 331

Wolfram Wingerath, Steffen Friedrich, Felix Gessert, Norbert Ritter
Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking 351

Query Optimization

Carl-Philip Hänsch, Thomas Kissinger, Dirk Habich, Wolfgang Lehner
Plan Operator Specialization using Reflective Compiler Techniques 363

Thomas Neumann, Alfons Kemper
Unnesting Arbitrary Queries 383

Yuanzhen Ji, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, Christof Fetzer
Optimization of Continuous Queries in Federated Database and Stream Processing Systems 403

Detection and Recognition

Meike Klettke, Uta Störl, Stefanie Scherzinger
Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores 425

Sebastian Kruse, Thorsten Papenbrock, Felix Naumann
Scaling Out the Discovery of Inclusion Dependencies 445

Magdalena Rischka, Stefan Conrad
Image Landmark Recognition with Hierarchical K-Means Tree 455

Thomas Böttcher, Ingo Schmitt
Erzeugung kalibrierter, metrischer Distanzen mittels multidimensionaler Skalierung 465

Big Data-Zentren – Vorstellung und Panel

Volker Markl, Erhard Rahm, Wolfgang Lehner, Michael Beigl, Thomas Seidl
Big Data-Zentren – Vorstellung und Panel 477

Dissertationspreise

Ulrike Fischer
Forecasting in Database Systems 483

Maximilian Dylla
An Overview on Querying and Learning in Temporal Probabilistic Databases 493

Industrieprogramm

Events, Processes, Analytics

Karin Büscher, Andreas Hausotter, Arne Koschel, Jan Seidel
Pragmatisches Business Activity Monitoring für Workflow Management-Systeme in einer SOA 507

Christoph Heinz
Event-driven tracking of important sections of business processes 527

Eric Peukert, Christian Wartner, Erhard Rahm
A Smart Link Infrastructure for Integrating and Analyzing Process Data 547

Knut Stolze, Josephine Rückert, Frank Butt
Integration of Spatial Data in Database Acceleration for Analytics 565

Systems and Applications

Uta Störl, Thomas Hauf, Meike Klettke, Stefanie Scherzinger
Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue? 579

Mazen Salous, Daniela Nicklas, André Bolles, Henning Mextorf
Semantic Data Exchange in e-Navigation 601

Stefan Mandl, Oleksandr Kozachuk, Markus Endres, Werner Kießling
Preference Analytics in EXASolution 613

Lars Butzmann, Stefan Klauck, Stephan Müller, Matthias Uflacker, Werner Sinzig, Hasso Plattner
Generic Business Simulation Using an In-Memory Column Store 633

Panel: „Big Data – Evolution oder Revolution in der Datenverarbeitung?“

Alexander Böhm, Mathias Golombek, Christoph Heinz, Henrik Löser, Alfred Schlaucher, Thomas Ruf
Panel: „Big Data – Evolution oder Revolution in der Datenverarbeitung?“ 647

Demoprogramm

Stefanie Scherzinger, Andreas Thor
AutoShard - a Java Object Mapper (not only) for Hot Spot Data Objects in NoSQL Data Stores 651

Nikhil-Kishor Rane, Omran Saleh
AutoStudio: A Generic Web Application for Transforming Dataflow Programs into Action 655

Stefanie Scherzinger, Meike Klettke, Uta Störl
Cleager: Eager Schema Evolution in NoSQL Document Stores 659

Michael Falkenthal, Johanna Barzen, Simon Dörner, Vadym Elkind, Jan Fauser, Frank Leymann, Tino Strehl
Datenanalyse in den Digital Humanities – Eine Annäherung an Kostümmuster mittels OLAP Cubes 663

Florian Stahl, Adrian Godde, Bastian Hagedorn, Bastian Köpcke, Martin Rehberger, Gottfried Vossen
High Quality Information Delivery: Demonstrating the Web in Your Pocket for Cineast Tourists 667

Robert Gottstein, Rohit Goyal, Ilia Petrov, Sergey Hardock, Alejandro Buchmann
MV-IDX: Multi-Version Index in Action 671

Till Kolditz, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner
Online Bit Flip Detection for In-Memory B-Trees Live! 675

Frank Eichinger, Immanuel Wietreich
RecoLeta: A Recommender System for Events for Personalised E-Mail Campaigns 679

Marwan Hassani, Christian Beecks, Daniel Töws, Tatiana Serbina, Max Haberstroh, Paula Niemietz, Sabina Jeschke, Stella Neumann, Thomas Seidl
Sequential Pattern Mining of Multimodal Streams in the Humanities 683

Johannes Schildgen, Stefan Deßloch
SQL-Grundlagen spielend lernen mit dem Text-Adventure SQL Island 687

Sandy Ganza, Thomas Psota, Daniel Schall, Theo Härder
Visualizing the Behavior of an Elastic, Energy-Efficient Database Cluster 691

Eingeladene Vorträge

Big Data: Hype and Reality

C. Mohan
cmohan@us.ibm.com

1 Abstract

Big Data has become a hot topic in the last few years in both industry and the research community. For the most part, these developments were initially triggered by the requirements of Web 2.0 companies. Both technical and non-technical issues have continued to fuel the rapid pace of developments in the Big Data space. Open source and non-traditional software entities have played key roles in the latter. As it always happens with any emerging technology, there is a fair amount of hype that accompanies the work being done in the name of Big Data. The set of clear-cut distinctions that were made initially between Big Data systems and traditional database management systems are being blurred as the needs of the broader set of (real world) users and developers have come into sharper focus in the last couple of years. In this talk, I will survey the developments in Big Data and try to distill reality from the hype!

2 Biography

Dr. C. Mohan has been an IBM researcher for 33 years in the information management area, impacting numerous IBM and non-IBM products, the research and academic communities, and standards, especially with his invention of the ARIES family of database locking and recovery algorithms, and the Presumed Abort commit protocol. This IBM, ACM and IEEE Fellow has also served as the IBM India Chief Scientist for 3 years. In addition to receiving the ACM SIGMOD Innovation Award (1996), the VLDB 10 Year Best Paper Award (1999) and numerous IBM awards, was elected to the US and Indian National Academies of Engineering, and was named an IBM Master Inventor. This Distinguished Alumnus of IIT Madras received his PhD at the University of Texas at Austin. He is an inventor of 42 patents. He has served on the advisory board of IEEE Spectrum and on the IBM Software Group Architecture Boards Council. Mohan is a frequent speaker in North America, Europe and India, and has given talks in 40 countries.

Novel Optimization Techniques for Modern Database Environments

Alexander Böhm
alexander.boehm@sap.com

1 Abstract

Performance and scalability have always been major aspects of the design and implementation of database management systems (DBMS). Consequentially, there is a vast number of publications addressing performance improvements for virtually every aspect of a DBMS, such as query processing, restart recovery, physical data storage, index structures, or buffer management, to only name a few.

In recent years, due to the evolution of hardware and its new capabilities (e.g. vector processing or hardware transactional memory), the topic of hardware/software co-design has gained much attention. Both DBMS researchers and developers have broadened their scope from a pure (system) software programming perspective towards a more hardware-aware view.

In this keynote, we will complement this trend of looking towards the hardware by considering another part of the usual software stack a DBMS is deployed in, and focus towards the application server running the user code. While the interaction and potential synergies between application server and DBMS are often neglected, we will discuss a multitude of optimization opportunities that can be addressed. Examples include, but are not limited to:

- Efficient data transfer between application server and DBMS
- Overhead reduction by data type and memory layout alignment
- Request batching (in the spirit of group commit)
- Code relocation (push-down and pull-up)

We discuss how these techniques allow to improve the overall efficiency of the software stack running the end user's applications and optimize their perceived performance and experience.

2 Biography

Alexander Böhm is a database architect working on SAP's HANA in-memory database management system with a strong focus on performance optimization. Prior to joining SAP, he received his PhD from the University of Mannheim, Germany where he worked on the development of efficient and scalable applications using declarative message processing.

When can “big data” be “in-memory”?

Eric Sedlar
eric.sedlar@oracle.com

1 Abstract

The monikers of “big data” and “in-memory” are certainly hyped in the database world, but some people might argue that they don’t overlap. The terms are vague enough to be treated as mutually exclusive as well as mostly overlapping. In addition, “big data” often implies “data science” which means “not SQL” (based on some programmable framework like Map-Reduce, Spark, or Flink). How do we see the “in-memory” and “big data” trends for analytics evolving in the future (separately or together) and what is the role of SQL vs. other frameworks?

2 Biography

Eric Sedlar is Vice President and Technical Director of Oracle Labs (formerly Sun Labs). This position entails figuring out how to transfer research results from Labs research into Oracle products and services, as well as setting overall technical direction for new research projects in Oracle Labs. Eric manages over 160 fulltime researchers working on over a dozen separate research areas.

Eric’s own research interests are in domain-specific languages and acceleration of database operations both via new hardware and using JIT compilation.

Previously, he led the effort for XML-native storage inside Oracle, starting with Oracle 9iR2. Eric has held various architecture and development management positions at Oracle since starting there in 1990. He holds over 68 patents, and has served on standards organizations for Oracle in the W3C and IETF. He co-authored the Best Paper at SIGMOD 2010 on architecture-sensitive search trees.

The Case for Small Data Management

Jens Dittrich, jens.dittrich@cs.uni-saarland.de

1 Abstract

Exabytes of data; several hundred thousand TPC-C transactions per second on a single computing core; scale-up to hundreds of cores and a dozen Terabytes of main memory; scale-out to thousands of nodes with close to Petabyte-sized main memories; and massively parallel query processing are a reality in data management. But, hold on a second: for how many users exactly? How many users do you know that really have to handle these kinds of massive datasets and extreme query workloads? On the other hand: how many users do you know that are fighting to handle relatively small datasets, say in the range of a few thousand to a few million rows per table? How come some of the most popular open source DBMS have hopelessly outdated optimizers producing inefficient query plans? How come people don't care and love it anyway? Could it be that most of the world's data management problems are actually quite small? How can we increase the impact of database research in areas when datasets are small? What are the typical problems? What does this mean for database research? We discuss research challenges, directions, and a concrete technical solution coined PDbF: Portable Database Files. This is an extended version of an abstract and Gong Show talk presented at CIDR 2015 (http://www.cidrdb.org/cidr2015/Papers/11_Abstract17DJ.pdf).

2 Biography

Jens Dittrich is a Full Professor of Computer Science in the area of Databases, Data Management, and Big Data at Saarland University, Germany. Previous affiliations include U Marburg, SAP AG, and ETH Zurich. He is also associated to CISPA (Center for IT-Security, Privacy and Accountability). He received an Outrageous Ideas and Vision Paper Award at CIDR 2011, a BMBF VIP Grant, a best paper award at VLDB 2014, two CS teaching awards in 2011 and 2013, as well as several presentation awards including a qualification for the interdisciplinary German science slam finals in 2012 and three presentation awards at CIDR (2011, 2013, and 2015).

His research focuses on fast access to big data including in particular: data analytics on large datasets, Hadoop MapReduce, main-memory databases, and database indexing. He has been a PC member and/or area chair of prestigious international database conferences such as PVLDB, SIGMOD, and ICDE. Since 2013 he has been teaching his classes on data management as flipped classrooms. See http://datenbankenlernen.de or http://youtube.com/jensdit for a list of freely available videos on database technology in German and English (about 80 videos in German and 80 in English so far).

Wissenschaftliches Programm

Cloud Data Management

KitMig – Flexible Live-Migration in mandantenfähigen Datenbanksystemen

Andreas Göbel

Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2
07743 Jena
andreas.goebel@uni-jena.de

Marcel Sufryd

Wirtschaftswissenschaftliche Fakultät
Friedrich-Schiller-Universität Jena
Carl-Zeiss-Straße 3
07743 Jena
marcel@sufryd.com

Abstract: Mandantenfähige Datenbanksysteme ermöglichen die gemeinsame Nutzung physischer Ressourcen durch eine Vielzahl von Mandanten. Ihr Einsatz erlaubt Anbietern von Cloud-Datenbankdiensten die Reduzierung der Betriebskosten durch eine hohe Ressourcennutzung und die Ausnutzung von Skaleneffekten. Die Migration von Mandanten innerhalb einer Serverfarm erweist sich in mandantenfähigen Datenbanksystemen als eine Schlüsselkomponente für Elastizität, Lastverteilung und Wartbarkeit. In diesen Einsatzbereichen werden jedoch unterschiedliche und zum Teil unvereinbare Anforderungen an eine Migration gestellt. Existierende Ansätze zur Live-Migration eignen sich aufgrund ihres statischen Ablaufs nur in wenigen Fällen.

In diesem Beitrag stellen wir das Framework KitMig zur Live-Migration in mandantenfähigen Datenbanksystemen vor. In Anlehnung an einen Baukasten stellt es verschiedene Module zur Bestimmung des Migrationsablaufs bereit. Die geeignete Kombination von Modulen erlaubt die Anpassung des Ablaufs an die gestellten Anforderungen. Im Rahmen des Beitrags werden die KitMig-Phasen, zugehörige Module und die Implementierung im Open-Source-DBMS H2[1] beschrieben. Mehrere Untersuchungen demonstrieren die Charakteristik verschiedener Modulkombinationen und die Anwendbarkeit der resultierenden Abläufe in ausgewählten Einsatzbereichen.

1 Einleitung

Relationale Datenbanksysteme sind zentraler Bestandteil des Software-Stacks vieler Unternehmen. Die Anbieter von Cloud-Datenbankdiensten (Database as a Service, DbaaS [SK12]) verfolgen das Ziel, Unternehmen eine Alternative zum Betrieb der Datenbanksysteme innerhalb des Unternehmens zu bieten. Angebote wie Windows Azure SQL-Datenbank[2] oder Amazon Relational Database Service (RDS)[3] sorgen beispielsweise für die Bereitstellung, Provisionierung, Konfiguration, Überwachung und Wartung von Datenbanken. Zu den Herausforderungen dieser Dienste gehört die adäquate Verwaltung einer großen Anzahl von Mandanten (Unternehmen, Organisationen oder Endkunden)

[1] http://www.h2database.com/
[2] http://www.windowsazure.com/services/sql-database/
[3] http://aws.amazon.com/rds/

mit vielfältigen und kaum vorhersehbaren Zugriffsmustern. Durch die Konsolidierung von Mandanten mit entgegengesetzten Aktivitätszeiträumen und Ressourcenanforderungen vermeiden Dienstanbieter die stetige Bereitstellung von Ressourcen für die maximale Last jedes einzelnen Mandanten [JA07]. Die Nutzung von Skaleneffekten als Folge höherer Ressourcenausnutzung und vereinfachter Wartung erlaubt ihnen die Reduzierung der Betriebskosten beim Einsatz dieser mandantenfähigen Datenbanksysteme. Die Anzahl der zu betreibenden und administrierenden Datenbanksysteminstanzen können durch sie deutlich reduziert werden. Ihre Einsatzbereiche sind nicht auf DbaaS beschränkt, sie sind beispielsweise ein entscheidendes Mittel bei der Datenverwaltung in den Bereichen Software-as-a-Service (SaaS) und Platform-as-a-Service (PaaS).

Die Implementierungsvarianten mandantenfähiger Datenbanksysteme verdeutlichen den Konflikt zwischen einem hohen Isolationsgrad der Mandantendaten und hoher Ressourcenausnutzung. Jacobs und Aulbach [JA07] unterscheiden drei wesentliche Konsolidierungsansätze:

- *Shared Machine:* Die Mandanten verwenden ein gemeinsames Datenbanksystem, besitzen jedoch separate Datenbanken.
- *Shared Process:* Die Mandanten teilen sich Datenbankprozesse, besitzen jedoch getrennte Tabellen.
- *Shared Table:* Die Daten mehrerer Mandanten liegen in gemeinsamen Tabellen.

Einige kommerzielle Systeme wie Amazon RDS sowie Forschungsprojekte wie Relational Cloud [CJP+11] setzen aufgrund der hohen Mandantenisolation auf den Ansatz *Shared Machine*. Die Isolation verursacht jedoch hohe Kosten aufgrund geringer Ressourcenauslastung und nicht realisierbarer mandantenübergreifender Administration. Dieser Effekt wird bei anknüpfenden Ansätzen wie separaten Datenbankinstanzen oder separaten virtuellen Maschinen (VMs) abermals verstärkt [Rei10]. Der Ansatz *Shared Table* erlaubt hingegen eine sehr effiziente Ressourcennutzung. Dienstanbieter müssen durch eine geeignete Abbildung die vielfältigen Mandantendaten in einem physischen Datenbankschema vereinen und zugleich die Isolation der Daten und der Performance von Mandanten gewährleisten [AJKS09]. Individualität bei der Administration und DBMS-Funktionalität kann hierbei in der Regel nicht geboten werden. Der Ansatz ist daher insbesondere bei Mandanten mit vergleichbaren Tabellenstrukturen, Zugriffsmustern und Anforderungsprofilen geeignet, was ihn beispielsweise für SaaS-Anbieter wie Salesforce.com [WB09] zur Datenverwaltung ihrer Dienste interessant macht. *Shared Process* stellt einen geeigneten Kompromiss aus Sicherheit, Performance und operativen Kosten dar und kommt aus diesem Grund beispielsweise bei Windows Azure SQL-Datenbank zum Einsatz.

Die dynamische Ressourcenverwaltung, Lastverteilung und Elastizität zählen zu den primären Anforderungen an mandantenfähige Datenbanksysteme [AJPK09, BZP+10]. Durch die verteilte Verarbeitung auf mehreren Rechnerknoten [Rah94] können sie eine Vielzahl von Mandanten adäquat unterstützen. Shared-Nothing-Architekturen weisen den Servern durch die Partitionierung des Datenbestands unabhängige Ressourcen zu, während die Server bei Shared-Disk-Architekturen einen zentralen Datenspeicher gemeinsam verwenden. Um auf Laständerungen adäquat reagieren zu können, bedürfen die Systeme geeigneter Platzierungs- und Migrationsstrategien [EDP+13, LHM+13, SJK+13, YQR+12] sowie

adäquater Migrationsverfahren [BCM+12, DNAEA11, EDAA11, SCM13] zur dynamischen Zuweisung der Mandanten zu Rechnerknoten. Dieser Beitrag konzentriert sich auf OLTP-Systeme, bei denen die Daten eines Mandanten nach Möglichkeit auf einen Server zu konzentrieren sind, um aufwändige verteilte Transaktionen zu vermeiden [CJZM10]. Migrationsstrategien legen zu migrierende Mandanten, Zielsysteme und Ausführungszeitpunkte der Migrationen fest. Sie überwachen den Ressourcenbedarf und die Performance von Mandanten und reagieren in Problemfällen, gegebenenfalls unter Einbeziehung von Lastvorhersagen auf Basis von Mandantenprofilen [CJP+11, EDP+13].

Im Rahmen dieses Beitrags zeigen wir vier Anforderungsprofile für die Live-Migration von Mandanten auf und ordnen ihnen Anforderungen an die Migrationsdurchführung zu. Wir kategorisieren Migrationsverfahren in drei Klassen und verdeutlichen deren Stärken und Schwächen. Zudem stellen wir das Framework KitMig zur flexiblen Live-Migration in relationalen Datenbanksystemen basierend auf dem Shared-Process-Ansatz vor. KitMig stellt verschiedene Module zur Verfügung, um den Migrationsablauf an vielfältige Anforderungen anzupassen und ist infolgedessen in unterschiedlichen Anwendungsgebieten verwendbar. Wir geben einen Überblick über die Implementierung KitMigs sowie über bereitgestellte Module und Verfahren. Gestützt durch die Ergebnisse verschiedener Untersuchungen leiten wir Einsatzbereiche ausgewählter KitMig-Verfahren ab.

Der Beitrag ist wie folgt aufgebaut: In *Kapitel 2* werden Bewertungskriterien und Anforderungsprofile von Migrationsverfahren aufgezeigt und Verfahrensklassen gegenübergestellt. *Kapitel 3* stellt das Framework KitMig vor und gibt einen Überblick über seine Implementierung und die bereitgestellten Module. *Kapitel 4* präsentiert die Ergebnisse verschiedener Untersuchungen. Es bewertet ausgewählte Migrationsverfahren und leitet ihre Eignung für die Anforderungsprofile ab. Im Rahmen des *Kapitels 5* werden veröffentlichte Verfahren zur Live-Migration vorgestellt und bewertet. Es geht zudem auf Forschungsgebiete mit vergleichbaren Anforderungen ein. *Kapitel 6* fasst die Ergebnisse des Beitrags zusammen und gibt einen Ausblick auf mögliche Erweiterungen und offene Fragestellungen.

2 Live-Migration

Als Migration wird das Übertragen eines Datenbestands von einem Quell- auf ein Zielsystem bezeichnet. Sie kann mit einem Wechsel der DBMS-Version oder des Betriebssystems verbunden sein und den Datenbestand etwa auf eine neue Festplatte innerhalb eines Servers, auf einen neuen Server innerhalb eines Rechenzentrums oder zu einem anderen Rechenzentrum übermitteln. Aufgrund ihrer großen Bedeutung in mandantenfähigen Datenbanksystemen ist sie nativ durch jene Systeme zu unterstützen [AJPK09, EDAA11].

Ein trivialer Migrationsansatz besteht in der Betriebsunterbrechung des Datenbanksystems und anschließender Erstellung, Übertragung und Wiederherstellung einer Sicherung. Ansätze wie dieses *Stop-And-Copy* [EDAEA11, SCM13] sind vor allem wegen nicht gewährleisteter Verfügbarkeit während der Migration für Cloud-Datenbankdienste nicht akzeptabel. Stattdessen sind Live-Migrationen gemäß *Definition 1* zu unterstützen.

Kategorie	Bewertungskriterium
Verfügbarkeit	Systemausfallzeit Ausfallzeit des Mandanten Fehlgeschlagene Mandantenanfragen Migration der Datenbankverbindung
Systembelastung	Mehrbelastung vor der Migration Mehrbelastung während der Migration Mehrbelastung nach der Migration Migrationsdauer Zeitpunkt der Lastmigration Kommunikationsaufwand
Koordination und Korrektheit	Autonomie Korrektheit Fehlertoleranz

Tabelle 1: Bewertungskriterien für Migrationsverfahren

► **Definition 1** *Als **Live-Migration** wird der Prozess innerhalb eines mandantenfähigen Datenbanksystems bezeichnet, der den gesamten Datenbestand eines Mandanten von einem Quell- auf ein Zielsystem übermittelt und dabei folgenden Kriterien genügt [EDAA11]:*

- *Keine Verursachung von Ausfallzeiten des Datenbanksystems,*
- *keine oder sehr geringe Beeinflussung der Erreichbarkeit der Daten des Mandanten,*
- *Verursachung keiner oder nur sehr weniger Transaktionsabbrüche des Mandanten,*
- *Übertragung bestehender DB-Verbindungen zum Zielsystem ohne Neuaufbau.*

2.1 Bewertungskriterien

Geeignete Metriken ermöglichen die Vergleichbarkeit verschiedener Migrationsverfahren. Sie werden in *Tabelle 1* aufgelistet, wobei die Kategorie Verfügbarkeit alle wesentlichen Merkmale einer Live-Migration gemäß *Definition 1* beinhaltet. Die weiteren Bewertungskriterien werden im Folgenden nach [BZP+10, EDAA11, EDAEA11] beschrieben.

Live-Migrationen führen insbesondere durch den Ressourcenbedarf ihrer Datenbewegungen unweigerlich zu einer Belastung des Quellsystems, Zielsystems und Netzwerks. Mehrbelastungen können sich als Erhöhung der Antwortzeit oder als Verringerung des Transaktionsdurchsatzes auswirken und sind für den zu migrierenden Mandanten und für zeitgleich aktive Mandanten messbar. Sie können bestehende Überlastungen verschärfen oder gar neue hervorrufen [BCM+12]. Notwendige Vorgänge im operativen Betrieb, welche die Voraussetzungen für Live-Migrationen schaffen, führen zu einer Systembelastung vor der Migration. Mehrbelastungen können zudem während der Migration auftreten. Der Verzicht auf das Übertragen des Pufferstatus von Datenbankseiten des migrierten Mandanten

(im Folgenden als Pufferinhalt bezeichnet) ist ein typisches Beispiel für eine Mehrbelastung, die aus einer Migration resultiert. Wichtige Einflussfaktoren auf die Systembelastung sind zudem die Migrationsdauer, der Übertragungszeitpunkt der Transaktionsverantwortung an das Zielsystem (Lastmigration) und der Kommunikationsumfang, der sowohl die übermittelten Mandantendaten als auch zusätzlich benötigte Inhalte umfasst.

Spezielle DBMS-Komponenten auf dem Quell- und Zielsystem sind für die Migrationsdurchführung und die korrekte Datenübertragung verantwortlich. Die Autonomie kennzeichnet den nötigen Koordinierungsaufwand zur Veranlassung und Durchführung der Migration. Während Korrektheit bei vollständiger Datenübertragung an das Zielsystem und ordnungsgemäßer Integration der Daten im Zielsystem gegeben ist, bewertet die Fehlertoleranz die transaktionale Korrektheit bei Ausfällen und in Fehlersituationen.

Nicht alle aufgeführten Metriken können von einem Migrationsverfahren gleichermaßen gut erfüllt werden. So steht beispielsweise eine geringe Systembelastung während der Migration in der Regel im Widerspruch zu einer geringen Migrationsdauer. Der Stellenwert jeder Metrik für eine Migration wird wesentlich durch das zugrunde liegende Anforderungsprofil bestimmt.

2.2 Anforderungsprofile

Die Live-Migration ist eine Schlüsselkomponente zur Optimierung der Ressourcenauslastung, indem sie die Zuweisung von Systemressourcen an Mandanten anpasst. Sie dient der Lastverteilungen, der Konsolidierung von Servern mit anhaltend geringer Auslastung (Serverkonsolidierung, Scale-In) und der Nutzung von Ressourcen hinzugefügter Server (Scale-Out). Lastverteilungen und Scale-Outs realisieren in der Regel Entlastungen von Quellsystemen mit (zu) hoher Last auf Zielsysteme mit geringer oder keinerlei Last, während ein Scale-In in der Regel bei (zu) geringer Last des Quellsystems durchgeführt wird und überlastete Systeme als Ziel meidet. Diese Unterscheidung führt in Verbindung mit den Ursachen einer Migration und der Dringlichkeit des Eingreifens zu den im Folgenden beschriebenen Anforderungsprofilen an Live-Migrationen.

Akute Entlastung: Diese Migrationen dienen der unverzüglichen Entlastung des Quellsystems und sind durch bestehende oder unmittelbar drohende finanzielle Aufwendungen für den Datenbankbetreiber aufgrund von Missständen auf dem Quellsystem motiviert. Zu den Hauptbeweggründen zählen Überlastungen des Quellsystems und Verletzungen der Service-Level-Agreements (SLAs) von Mandanten infolge von unvorhersehbarem Nutzungsverhalten. Zur schnellen Entlastung des Quellsystems ist eine möglichst frühe Lastmigration vonnöten. Zudem sind Mehrbelastungen auf dem Quellsystem und aus Migrationen resultierende Mehrbelastungen zu minimieren, um die Situation nicht weiter zu verschärfen.

Vorausplanende Entlastung: Dieses Profil umfasst Migrationen mit dem Ziel der Entlastung des Quellsystems zur Vermeidung zukünftiger Überlastungen. Zu ihren Ursachen zählen beispielsweise eine serverübergreifende Lastbalancierung, SLA-Änderungen oder erwartete SLA-Verletzungen. Zudem können sie durch erwartete Last-

anstiege, proaktiv hinzugefügte Server oder eine höhere Anzahl zu verwaltender Mandanten hervorgerufen werden. Sie bedürfen einer frühen Lastmigration sowie einer geringen Mehrbelastung auf dem Quellsystem und nach der Migration.

Akute Serverkonsolidierung: Unaufschiebbare Wartungsarbeiten stellen die Hauptursache für eine umgehende Deaktivierung von Datenbankservern mit dem Ziel einer akuten Serverkonsolidierung dar. Sie werden beispielsweise durch Fehlermeldungen sich selbst überwachender Hardware ausgelöst und bedürfen der umgehenden Migration aller durch den Server verwalteten Mandanten. Die Migrationen sollten eine minimale Laufzeit anstreben und Mehrbelastungen auf dem Zielsystem minimieren, um dessen Überlastung zu vermeiden. Mehrbelastungen auf dem Quellsystem und eine späte Lastmigration können hingegen in Kauf genommen werden.

Vorausplanende Serverkonsolidierung: Migrationen dieses Profils führen zeitunkritische Serverkonsolidierungen durch. Mögliche Ursachen bestehen in geplanten Wartungsarbeiten oder der Entfernung von Servern mit geringer Last aus Kostengründen. Eine reduzierte Last kann beispielsweise durch zurückgehende Ressourcenanforderungen der Mandanten oder einer Reduzierung der Mandantenanzahl hervorgerufen werden. Analog zu akuten Serverkonsolidierungen sind hierbei Mehrbelastungen für das Zielsystem zu vermeiden. Eine relativ lange Migrationsdauer ist bei entsprechendem Planungshorizont hingegen tolerierbar.

2.3 Verfahrensklassen

Verfahren zur Live-Migration in mandantenfähigen Datenbanksystemen müssen folgende Kernprozesse realisieren:

Datenmigration: Dieser Prozess ist für die Übertragung des kompletten Datenbestands des zu migrierenden Mandanten vom Quell- auf das Zielsystem verantwortlich. Die Daten können durch das Quellsystem bereitgestellt (Push-Strategie) oder durch das Zielsystem bedarfsgerecht angefordert (Pull-Strategie) werden. In Shared-Disk-Architekturen sind ausschließlich Arbeitsspeicherinhalte wie Pufferinhalte des Mandanten zu übertragen, während Shared-Nothing-Architekturen die Übertragung persistenter Daten erfordern und die Migration der Pufferinhalte optional ist.

Lastmigration: Sie leitet bestehende Datenbankverbindungen des Mandanten vom Quellsystem auf das Zielsystem um. Laufende Transaktionen auf dem Quellsystem sind abzubrechen, ebenfalls zu migrieren oder weiterhin auf dem Quellsystem auszuführen und mit neuen Transaktionen auf dem Zielsystem zu synchronisieren.

Bereinigung: Nach erfolgter Daten- und Lastmigration ist der Datenbestand des migrierten Mandanten auf dem Quellsystem vollständig zu entfernen.

Migrationsverfahren variieren bezüglich der Prozessabfolge und -implementierung. Dabei ist zwischen reaktiven (post-copy), proaktiven (pre-copy) und hybriden Verfahren zu unterscheiden [AEDE11, SCM13].

Reaktive Verfahren [EDAEA11] stellen dem Zielsystem zu Migrationsbeginn die zur Verarbeitung von Transaktionen des Mandanten notwendigen Kataloginhalte bereit. Anschließend führen sie die Lastmigration durch, woraufhin das Zielsystem im Rahmen der Datenmigration die zur Transaktionsverarbeitung notwendigen Daten vom Quellsystem anfordert. Dieser Vorgang wird durch parallele Bereitstellung der restlichen Daten durch das Quellsystem begleitet. Die Verfahren erreichen eine schnelle Entlastung des Quellsystems. Die bedarfsgerechte Anforderung führt jedoch zu einem großem Kommunikationsaufwand, einer langen Migrationsdauer und einer hohen Belastung während der Migration. Reaktive Verfahren eignen sich daher insbesondere für die akute und vorausplanende Entlastung.

In *proaktiven Verfahren* [BCM+12, DNAA10, DNAEA11] stellt das Quellsystem dem Zielsystem bereits vor der Lastmigration den kompletten Datenbestand bereit. Durch die gebündelte Übertragung des Datenbestands halten sie den Kommunikationsaufwand und die Mehrbelastungen während der Migration gering und reduzieren die Migrationsdauer. Jedoch entlasten sie durch die lange Datenübertragungsdauer vor der Lastmigration das Quellsystem sehr spät. Demzufolge eignen sich proaktive Verfahren vorrangig bei akuten und geplanten Serverkonsolidierungen.

Hybride Verfahren [SCM13] verbinden die vorherigen Ansätze durch die verschränkte Ausführung der Daten- und Lastmigration sowie die Kombination von Datenbereitstellungen und -anforderungen. Dies erlaubt insbesondere die Bereitstellung häufig verwendeter Mandantendaten (Hot Pages) vor der Lastmigration, während nach ihr die weiteren Daten (Cold Pages) bereitgestellt oder bedarfsgerecht angefordert werden. Die Bereitstellung von Hot Pages dient der Reduzierung bedarfsgerechter Seitenanforderungen. Der Anteil der Hot Pages am Gesamtdatenbestand bestimmt die Charakteristik der Migration, welche sich zwischen proaktiven Verfahren (ausschließlich Hot Pages) und reaktiven Verfahren (keine Hot Pages) bewegt.

Die Bewertung der Verfahrensklassen und die Ableitung ihrer Eignung für die verschiedenartigen Anforderungsprofile von Live-Migrationen unterstreichen, dass kein universelles Migrationsverfahren existieren kann, das in allen Anwendungsgebieten uneingeschränkt verwendbar ist. Hybride Verfahren stellen einen geeigneten Ansatzpunkt hierzu dar, sie müssen jedoch Freiheitsgrade zur Abstimmung ihres Ablaufes auf den jeweiligen Anwendungsfall bereitstellen.

3 KitMig

Mit der Entwicklung von KitMig verfolgten wir das Ziel, verschiedene Migrationsabläufe in einem Migrationsverfahren für mandantenfähige Datenbanksysteme zu vereinen. Diese Flexibilität erlaubt dem DBMS die Festlegung des geeigneten Migrationsablaufs in Abhängigkeit vom gegebenen Anforderungsprofil und dem Systemzustand. Die folgenden Abschnitte erläutern die Systemarchitektur, stellen die Migrationsphasen und zugehörige Module KitMigs vor und gehen auf die Implementierung und bestehende Einschränkungen ein. Eine umfassende Beschreibung des Frameworks ist in [Göb15] zu finden.

3.1 Systemarchitektur und Design-Entscheidungen

KitMig setzt einen Datenbank-Cluster mit Shared-Nothing-Architektur voraus. Jeder Datenbankserver verwaltet dabei die Daten mehrerer Mandanten, wobei KitMig die maximale Anzahl der Mandanten je Server nicht einschränkt. Gemäß dem Konsolidierungsansatz *Shared Process* wird jedem Mandanten ein Datenbankschema zur Speicherung seiner Datenbankobjekte zugewiesen. Im Vergleich zu *Shared Machine* ruft dieser Ansatz zusätzlichen Herausforderungen für Live-Migrationen wie die gemeinsame Nutzung des Datenbankpuffers und der Datenbankdateien sowie potenzielle Konflikte der Seiten-IDs bei ihrer Einbindung auf dem Zielsystem hervor, mit denen KitMig umgeht. Der Einsatz KitMigs für *Shared Machine* bedarf nur geringfügiger Anpassungen. Der Ansatz *Shared Table* bringt hingegen weitere Herausforderungen mit sich, so schließt beispielsweise die Speicherung der Daten mehrerer Mandanten innerhalb einer Datenbankseite den Einsatz KitMigs aus. Aufgrund seiner Konzipierung für OLTP-Systeme unterstellt KitMig Transaktionen mit kurzer Laufzeit und wenigen Zugriffen auf Datenbankseiten. Bereichsanfragen wirken sich in KitMig negativ auf die Mehrbelastung während der Migration, den Kommunikationsaufwand und die Migrationsdauer aus. KitMig setzt zudem voraus, dass die Ressourcen eines Datenbankservers zur adäquaten Verwaltung aller Daten eines Mandanten genügen und keine knotenübergreifende Partitionierung angewendet wird, was einen gebräuchlichen Ansatz [YSY09] darstellt.

Unter den gegebenen Voraussetzungen umfasst die Live-Migration eines Mandanten bei KitMig die Übertragung aller Inhalte eines Datenbankschemas von einem Quell- an einen Zielserver. Ein Server kann als Quelle und Ziel mehrerer paralleler Migrationen agieren. Die Migration wird über eine SQL-Erweiterung durch Administratoren oder interne DBMS-Komponenten wie Controller zur Realisierung von Migrationsstrategien ausgelöst. Die Lastmigration realisiert KitMig durch einen externen Proxy-Server, der horizontale Skalierbarkeit bietet und eingehende Anfragen basierend auf dem Standardschema der Datenbankverbindung an den zuständigen Datenbankserver umleitet. KitMig passt die Umleitungsvorschriften im Rahmen einer Migration an.

3.2 Migrationsphasen

Abbildung 1 verdeutlicht den Migrationsablauf bei KitMig, der in vier Phasen erfolgt: Datenbereitstellung durch den Quellserver, Synchronisierung paralleler Transaktionsverarbeitung (Dualbetrieb), Datenanforderung durch den Zielserver und Bereinigung des Quellservers. Aufgrund seines Ablaufs ist KitMig als hybrides Migrationsverfahren einzuordnen. Es folgt eine kurze Erläuterung der Funktionen aller Migrationsphasen.

Die *Bereitstellungsphase* erlaubt dem Zielserver die Verarbeitung von Transaktionen in zukünftigen Phasen. Dies wird durch die Übertragung eines Schema-Grundgerüsts ermöglicht, welches die Schema- und Datendefinitionen zu Tabellen, Indizes, Views, Sequenzen etc. sowie die Indexstrukturen ohne jegliche Mandantendaten enthält. Zudem erlaubt KitMig die Bereitstellung einer beliebige Menge der im Puffer befindlichen Mandantendaten.

Die *Synchronisationsphase* dient zur Lastmigration und Abarbeitung der am Quellserver

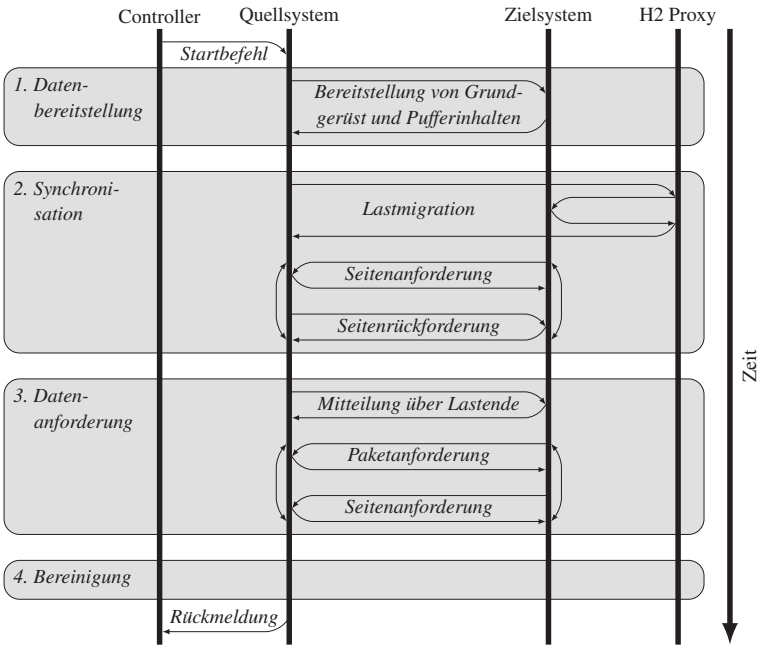


Abbildung 1: Ablauf einer Migration mit KitMig

noch ausstehenden Transaktionen, während neu eingehende Anfragen bereits durch den Proxy zum Zielserver umgeleitet werden. Der exklusive Datenzugriff zwischen den Servern wird durch ein Eigentumskonzept der Mandantendaten geregelt. KitMig ordnet Datenbankseiten einen Eigentümer zu und ermöglicht den Änderung des Eigentümers durch die Anforderung und Rückforderung des Seiteneigentums und -inhalts.

Die *Anforderungsphase* beginnt nach der Verarbeitung der letzten am Quellserver aktiven Transaktion. Zu diesem Zeitpunkt liegen die Mandantendaten teilweise bereits am Zielserver vor. Er fordert alle im Besitz des Quellservers befindlichen Datenbankseiten an. Für laufende Transaktionen benötigte Daten werden analog zur Synchronisationsphase seitenweise angefordert, während die weiteren Daten in Paketen übertragen werden.

Die *Bereinigungsphase* dient der abschließenden Entfernung des migrierten Datenbankschemas am Quellserver und der Migrationsmetadaten auf beiden Servern.

3.3 Vergleich bereitgestellter Module

KitMig stellt für die Phasen 1 – 3 verschiedene Module und Parameter zur individuellen Anpassung der Migrationsdurchführung an das Anforderungsprofil bereit. Dieser Abschnitt stellt die Module jeder Phase gegenüber. Das Modul `PUSH_WIREFRAME` stellt dem Zielserver das Schema-Grundgerüst bereit. Es realisiert ein reaktives Verfahren zur schnellen Entlastung des Quellservers, ist jedoch mit langen Zugriffszeiten aufgrund von vielen

Modul in	Eigentumsrecht		Seiteninhalt	
Synchronisationsphase	**QS ⇒ ZS**	**QS ⇐ ZS**	**QS ⇒ ZS**	**QS ⇐ ZS**
SIMPLEX	+	–	+	–
DUPLEX_UNMODIFIED	+	+/–	+	–
DUPLEX_ALL	+	+	+	+

QS: Quellserver ZS: Zielserver +: Zulässiger Transfer –: Unzulässiger Transfer

Tabelle 2: Gegenüberstellung der Übertragungsvorschriften zur Synchronisation

Seitenanforderungen in den Folgephasen verbunden. Im Vergleich dazu erzielt PUSH_BUFFER durch das Übertragen von Pufferinhalten kürzere Zugriffszeiten auf die meistverwendeten Mandantendaten auf Kosten einer späteren Lastmigration. Die Lastsituation auf dem Quellserver und der Aktivitätsgrad des Mandanten bestimmen hierbei die Charakteristik der Migration, welche sich zwischen proaktiven Verfahren (vollständige Pufferung der Mandantendaten) und reaktiven Verfahren (keine Pufferinhalte des Mandanten) bewegt.

Die Module der Synchronisationsphase ermöglichen die Abwägung zwischen einer Vielzahl abgebrochener Transaktionen und hohem Kommunikationsaufwand. *Tabelle 2* verdeutlicht, dass alle Module die Übertragung von Eigentumsrechten und Seiteninhalten zum Zielserver gewähren. Das Modul SIMPLEX erlaubt ausschließlich die genau einmalige Übertragung von Daten zum Zielserver und bietet somit die geringste Netzwerkbelastung. Transaktionen auf dem Quellserver, die Zugriff auf bereits übertragene Daten benötigen, müssen jedoch abgebrochen werden. DUPLEX_UNMODIFIED erlaubt die Übertragung des Eigentumsrechts unveränderter Daten zurück zum Quellserver. Das liberalste Modul DUPLEX_ALL bringt dagegen keine Einschränkung von Seitenübertragungen mit sich und verursacht somit keine Transaktionsabbrüche aufgrund von Eigentumskonflikten.

KitMig stellt zwei Module für die Anforderungsphase bereit. Das Modul PULL_AND_BUFFER dient der Erwärmung des Datenbankpuffers auf dem Zielserver. Im Rahmen von Seitenpaketen übermittelte Datenbankseiten fügt er in den Datenbankpuffer ein, falls sie sich im Datenbankpuffer des Quellservers befanden. Als Beispielszenario sei hier die akute Entlastung eines sehr aktiven Mandanten genannt, die eine schnelle Entlastung des Quellservers fordert und daher unter Umständen nicht alle zu migrierenden Pufferdaten bereits vor der Lastmigration übertragen kann. Alternativ kann PULL_ONLY gewählt werden, um die verbleibenden Daten ohne Übernahme des Pufferzustands zu migrieren.

3.4 Implementierung

KitMig wurde prototypisch in das Open-Source-DBMS H2 (Version 1.3.171) integriert, da es alle benötigten Funktionen wie Isolation auf Schema-Ebene, Multithreaded-Sitzungen und baumbasierte Primär- und Suchindextypen mitbringt. Zudem bietet es einen übersichtlich strukturierten und zugleich überschaubaren Programmcode. Lediglich die auf Tabel-

lenebene erfolgende Sperrverwaltung in H2 führte zu Herausforderungen bei der Realisierung KitMigs. Dessen Konzeption ist nicht auf H2 zugeschnitten, sondern lässt eine Integration in andere DBMS zu.

Der SQL-Befehlssatz von H2 wurde um Befehle für die Initiierung der Migration (inklusive der Module und Parameterwerte), die Anpassung des Ablaufs laufender Migrationen und die Server-zu-Server-Kommunikation erweitert. Am Quellserver wird der Migrationsablauf durch einen Controller-Thread koordiniert. Zudem verfügt jede Datenbank über einen Migrations-Manager, der Metadaten wie Authentifizierungsdaten und Serveradressen zu laufenden Migrationen verwaltet. Um die Unveränderlichkeit des Schema-Grundgerüsts zu erzwingen, setzt KitMig zu Beginn der Migration eine Strukturspere, die Veränderungen an den Datendefinitionen und Indexstrukturen während der Initialisierungs- und Synchronisationsphasen verbietet. Sie unterbindet die Ausführung von DDL-Anweisungen zur Erstellung, Änderung und Entfernung von Datenbankobjekten des Schemas sowie DML-Anweisungen, die zu Änderungen der Indexstrukturen führen. Dies betrifft einen Teil der `UPDATE`-Befehle sowie alle `INSERT`- und `DELETE`-Befehle.

Anschließend erstellt KitMig das Schema-Grundgerüst durch die Serialisierung der Suchindizes sowie der inneren Knoten der Primärindizes. Beim Modul `PUSH_BUFFER` werden zusätzlich die im Datenbankpuffer des Quellservers befindlichen Seiten des Mandanten in das Schema-Grundgerüst integriert. Zur Verhinderung einer späten Lastmigration sehr aktiver Mandanten erlaubt KitMig die Einschränkung der zu migrierenden Pufferinhalte durch einen Parameter. Auf die Erzeugung dieses Grundgerüsts folgt seine Übertragung zum Zielserver als LOB-Parameter innerhalb eines SQL-Befehls. Die Daten werden auf dem Zielserver deserialisiert und in die lokale Datenbank eingebunden. Bei der Deserialisierung von Seitendaten wird zur Vorbeugung potenzieller Konflikte der Seiten-IDs auf dem Quell- und Zielserver jeder einzubindenden Seite eine neue ID zugeteilt und die Zuordnung der IDs des Quell- und Zielservers für die spätere Kommunikation zwischen den Servern aufbewahrt. Die Strukturspere gewährleistet die Konsistenz dieser Zuordnung während der Migration.

Das Eigentumskonzept ermöglicht exklusive Seitenzugriffe auf Blattknoten der Primärindizes während der Migration. Der Zielserver erhält zu Beginn der Synchronisationsphase das Eigentum an den migrierten Pufferinhalten, während alle weiteren Blattknoten dem Quellserver zugewiesen werden. Seitenzugriffe auf dem Quell- und Zielserver sind stets mit einer Vorabprüfung des Seiteneigentums verbunden. Besitzt der Server die Seite nicht, so ist sie vom anderen Server anzufordern. Das verwendete Modul der Synchronisationsphase entscheidet über die Zulassung der Eigentumsübertragung. Nicht zulässige Seitenanforderungen führen zum Abbruch der zugehörigen Transaktion. Zur Realisierung des Moduls `DUPLEX_UNMODIFIED` überwacht KitMig Änderungen übertragener Datenbankseiten. Diese Überwachung erlaubt KitMig zudem die Vermeidung unnötiger Übertragungen von Seiteninhalten, indem es bei Anforderungen unveränderter Datenbankseiten lediglich das Eigentumsrecht überträgt und auf die Übertragung des Seiteninhalts verzichtet. Seiteninhalte sind folglich nur nach Änderungen wiederholt zu übertragen, was minimalen Netzwerk-Overhead bei lesenden Anfragen bedeutet. Während der Synchronisationsphase bestimmt KitMig regelmäßig die Anzahl der auf dem Schema aktiven Sitzungen. Sobald alle Sitzungen geschlossen wurden, endet die Synchronisationsphase und

der Quellserver signalisiert dem Zielserver den gemeinsamen Wechsel in die Anforderungsphase. Durch einen optionalen Parameter kann die Dauer der aufwendigen Synchronisationsphase begrenzt werden, indem nach Ablauf der maximalen Laufzeit langlaufende Transaktionen gezielt abgebrochen werden.

In der Anforderungsphase ermittelt der Zielserver für alle Tabellen die noch zu übertragenden Datenbankseiten und fordert sie in Paketform an. Die Übertragung und Einbindung der Seiten beeinflusst sowohl das Netzwerk als auch die Transaktionsverarbeitung auf dem Zielserver. Zur Regulierung der Belastung sind zu Migrationsbeginn die Größe der Pakete und die Wartezeit zwischen den Paketanforderungen festzulegen. KitMig erlaubt zudem die Anpassung beider Werte während des Migrationsverlaufs und eine dynamische Drosselung der zu verwendenden Netzwerkbandbreite für die Paketübertragung, um die Mehrbelastungen in dieser Phase zu kontrollieren.

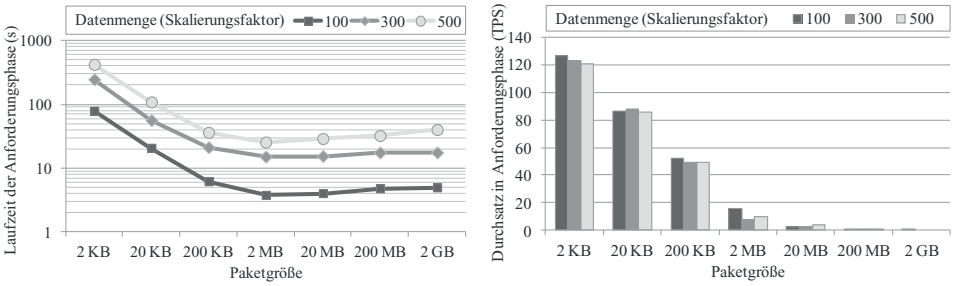
3.5 Einschränkungen

KitMig erlaubt die Migration persistenter Standard-Indextypen. Nicht unterstützt wird die Migration von reinen In-Memory-Indizes und Large Objects. Die Daten eines Mandanten sind zudem vollständig zu übertragen, eine Aus- oder Abwahl einzelner Tabellen ist nicht möglich. KitMig protokolliert alle Informationen auf dem Quell- und Zielserver, die zur Wiederherstellung des Systemzustands und Migrationsstatus nach einem Systemausfall auf einem oder beiden Servern notwendig sind. Die angelegten Dateien beinhalten den Migrationsablauf, die Zugangsdaten der beteiligten Server, die aktuelle Migrationsphase, das Mapping von Seiten-IDs und die Eigentumsrechte. Die Live-Migration verursacht zusätzliche Herausforderungen für eine log-basierte Systemwiederherstellung, deren Realisierung zu unseren zukünftigen Arbeiten gehört.

4 Experimentelle Untersuchungen

Dieses Kapitel stellt nach einem Überblick der Testumgebung die Ergebnisse zweier ausgewählter Untersuchungen des KitMig-Frameworks vor. Bei den Untersuchungen kamen vier baugleiche Server zum Einsatz, die über 1 Gigabit Ethernet miteinander verbunden sind. Sie besitzen jeweils eine 3.4 GHz Intel Core i7-2600 CPU (4 Kerne), 8 GB RAM, eine Festplatte mit 7200 RPM und werden durch Debian 4.7.2-4 (64 Bit) betrieben. Zwei Server agierten als Datenbankserver mit einem um KitMig erweiterten H2 basierend auf der H2-Version 1.3.171. Jeder Datenbankserver verwaltet die Daten mehrerer Mandanten in einer gemeinsamen Datenbank durch Zuweisung separater Schemata. Der im Rahmen des Projekts entwickelte Datenbank-Proxy *H2 Proxy* [Göb14a] wurde auf einem separaten Proxy-Server betrieben. Er nimmt alle Anweisungen entgegen und leitet sie gemäß dem verwendeten Schema an den zuständigen Datenbankserver weiter (siehe Abschnitt 3.1).

Der vierte Server diente der Daten- und Lastgenerierung durch das für mandantenfähige Datenbanksysteme entwickelte Benchmark-Framework *MuTeBench* [Göb14b]. Wir verwendeten es für Untersuchungen basierend auf dem Yahoo! Cloud Serving Benchmark



(a) Laufzeit der Anforderungsphase

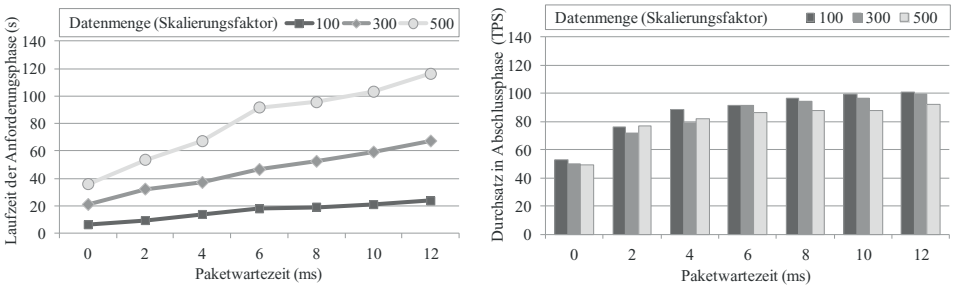
(b) Durchsatz in der Anforderungsphase

Abbildung 2: Einfluss der Paketgröße auf die Anforderungsphase (ohne Paketwartezeit)

(YCSB) [CST+10]. Im Gegensatz zu klassischen Datenbank-Benchmarks wurde dieser zur Evaluation von Cloud-Datenbanken entwickelt. Hierzu stellt er mehrere Lastszenarien mit variierendem Anteil lesender Anfragen bereit und bietet unterschiedliche Zugriffsmuster wie Gleichverteilung oder Zipf-Verteilung. Wir erweiterten YCSB um neue Lastszenarien und verwendeten in den Untersuchungen einen Transaktionsmix mit 70 % `READ`, 10 % `UPDATE`, 10 % `INSERT` und 10 % `DELETE` bei jeweils 10 Anweisungen pro Transaktion und Verwendung der Zipf-Verteilung. Jeder Mandant besitzt genau eine Tabelle mit 11 Spalten und 100.000 Zeilen (YCSB-Skalierungsfaktor 100, ca. 200 MB), 300.000 Zeilen (Faktor 300, ca. 610 MB) oder 500.000 Zeilen (Faktor 500, ca. 990 MB). Wir verwendeten eine Seitengröße von 2 KB, eine Puffergröße von 5 GB und das Isolationslevel Read Committed.

4.1 Paketgröße und -wartezeit

Als Grundlage für den Vergleich von KitMig-Verfahren sei auf den Einfluss verschiedener Parameter auf die Anforderungsphase eingegangen. Der effizienten Datenübertragung kommt in dieser Phase eine besondere Bedeutung zu, da sie insbesondere bei der Wahl von `PUSH_WIREFRAME` oder bei einem geringen Anteil der bereitgestellten Pufferinhalte an den zu migrierenden Daten den Großteil der Migrationsdauer ausmacht. Die Mandantendaten wurden mittels einer H2-Funktion komplett in den Puffer geladen, um Nebeneffekte aufgrund von Festspeicherzugriffen zu reduzieren. Es kam das KitMig-Verfahren `PUSH_WIREFRAME`, `SIMPLEX` und `PULL_ONLY` zum Einsatz. Beim ersten Test verzichteten wir auf Paketwartezeiten und nutzten Paketgrößen von einer Seite (2 KB) bis zu einer Million Seiten (2 GB). Mit zunehmender Paketgröße verringert sich der Kommunikationsaufwand pro übertragener Seite. Zudem kann die Effizienz der Seitenerfassung auf dem Quellserver und der Seitenintegration auf dem Zielserver erhöht werden und der Zielserver fordert weniger Seiten zur Einzelübertragung an. *Abbildung 2a* zeigt die resultierende Verringerung der Laufzeit der Anforderungsphase. Ab Paketgrößen von 20 MB wird nicht mehr von paralleler Serialisierung und Paketübermittlung profitiert, weshalb die



(a) Laufzeit der Anforderungsphase

(b) Durchsatz in der Anforderungsphase

Abbildung 3: Einfluss der Paketwartezeit auf die Anforderungsphase (Paketgröße: 200 KB)

Laufzeiten ansteigen. Sperren bei der Integration schränken die Verfügbarkeit der Daten während der Deserialisierung ein. *Abbildung 2b* zeigt, dass durch die längere Sperrdauer bei zunehmender Paketgröße folgerichtig die Transaktionsverarbeitung auf dem Zielserver deutlich beeinflusst oder gar verhindert wird.

Die Paketgröße von 200 KB stellt einen Kompromiss aus passabler Laufzeit und hohem Durchsatz dar. Wir verwendeten sie bei der Untersuchung von Paketwartezeiten. Der Zielserver wartete in den Untersuchungen bis zu 12 Millisekunden zwischen der Deserialisierung der Seiten eines Pakets und der Anforderung des folgenden Pakets. *Abbildung 3a* verdeutlicht den nahezu konstanten Anstieg der Laufzeit bei zunehmender Wartezeit. Gemäß *Abbildung 3b* ist bis 6 ms Wartezeit ein deutlicher Anstieg des Durchsatzes erkennbar. Daher kann mit 200 KB-Paketen und 4 ms Wartezeit bei deutlich geringerer Laufzeit ein ähnlicher Durchsatz erzielt werden wie bei einer Paketgröße von 20 KB ohne Wartezeit. Die *Abbildungen 2b* und *3b* verdeutlichen, dass der erzielte Durchsatz in beiden Untersuchungen durch größere Datenmengen nur geringfügig beeinflusst wird, was insbesondere auf die Pufferung der gesamten Datenmenge zurückzuführen ist.

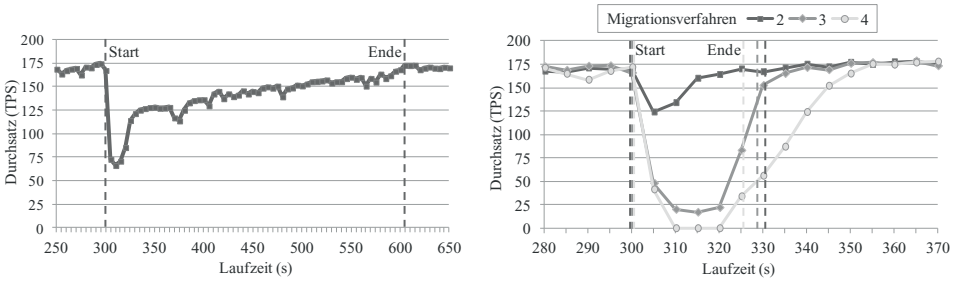
Wir führten zudem Untersuchungen mit statischer und dynamischer Drosselung der Übertragungsgeschwindigkeit durch. Sie ist ein geeignetes Mittel zur Reduzierung der Systemlast. Ihre regelmäßige Kontrolle der zu übertragenen Datenmenge bedingt jedoch einen erhöhten Aufwand, weshalb wir uns für die folgende Untersuchung auf feste Wartezeiten zwischen dem Versand von Paketen beschränkten.

4.2 Gegenüberstellung ausgewählter Verfahren

Aufgrund der Vielzahl möglicher Kombinationen von Modulen und Parameterwerten ist eine Untersuchung aller denkbaren Verfahren nicht realisierbar. Zur Gegenüberstellung von KitMig-Verfahren wählten wir gezielt vier Verfahren aus, deren Module und Parameterwerte in *Tabelle 3* zusammengefasst sind. Die Verfahren zeigen die Bandbreite der Möglichkeiten KitMigs auf. Die Daten des zu migrierenden Mandanten (Skalierungsfaktor 300, ca. 610 MB) wurden bei dieser Untersuchung nicht vollständig gepuffert, der Quellserver jedoch zur Puffererwärmung vor der Migration einer 5-minütigen Last ausgesetzt.

Nr.	Modul in Bereitstellungsphase	Modul in Synchronisationsphase	Modul in Anforderungsphase	Paket-größe	Paket-wartezeit
1	PUSH_WIREFRAME	DUPLEX_UNMODIFIED	PULL_AND_BUFFER	20 KB	4 ms
2	PUSH_BUFFER	DUPLEX_ALL	PULL_ONLY	200 KB	4 ms
3	PUSH_WIREFRAME	SIMPLEX	PULL_AND_BUFFER	2 MB	2 ms
4	PUSH_WIREFRAME	SIMPLEX	PULL_ONLY	2 GB	0 ms

Tabelle 3: Module und Parameterwerte der untersuchten KitMig-Verfahren



(a) Durchsatz für Verfahren 1

(b) Durchsatz für Verfahren 2 – 4

Abbildung 4: Erzielter Durchsatz der untersuchten KitMig-Verfahren

Verfahren 1 verfolgt das Ziel einer geringen Quellserverbelastung und ist insbesondere für die vorausplanende Entlastung und mit Einschränkungen für die akute Entlastung geeignet. Nach der Übertragung des Schema-Grundgerüsts mit lediglich 74 abgebrochenen Transaktionen (Abbruchquote von <0,1 % über den gesamten Migrationszeitraum) kann hierbei die Transaktionsverantwortung im Rahmen einer kurzen Synchronisationsphase schnell auf den Zielserver übertragen werden. Bei der Festlegung der Paketgröße und Paketwartezeit wurde das Ziel einer geringen Belastung während der Migration verfolgt. Die gewählten Werte führen zu einer ausgedehnten Anforderungsphase (siehe *Abbildung 5*), in der der Quellserver die Daten zur Entlastung nur langsam an den Zielserver übermittelt. In der Folge werden beträchtliche 24 % der Seiten vom Zielserver nach Bedarf angefordert und der Durchsatz des migrierten Mandanten nähert sich nur langsam dem Optimum an (siehe *Abbildung 4a*). Bei Migrationen von Mandanten mit Performance-Problemen sind daher andere Parameterwerte zu wählen.

Verfahren 2 überträgt in der Bereitstellungsphase die kompletten Pufferinhalte des Mandanten (58 % des Gesamtvolumens) zum Zielserver. In der aktuellen KitMig-Version bedarf es hierzu einer Strukturspere, die zu 525 Transaktionsabbrüchen (Abbruchquote von ca. 1,4 % über den gesamten Migrationszeitraum) führt und die Verfügbarkeit somit in Form von Dienstunterbrechungen einschränkt. Nach einer kurzen Synchronisationsphase werden die verbliebenen Daten angefordert. Im Vergleich zu Verfahren 1 sorgen sowohl die erhöhte Paketgröße als auch das erheblich verringerte Datenvolumen der anzufordernden Daten zu einer deutlich schnelleren Übertragung verbliebener Seiten (siehe *Abbildung 4b*). Der Zielserver fordert daher nur 6 % der Seiten nach Bedarf an. Das Verfahren eignet

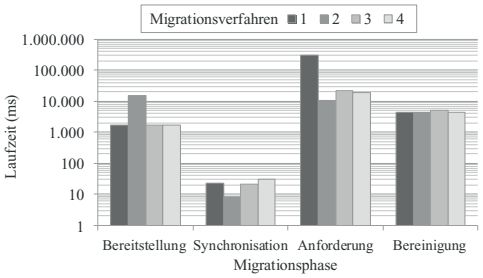


Abbildung 5: Laufzeit der untersuchten KitMig-Verfahren

sich für vorausplanende Serverkonsolidierungen und ist mit Einschränkungen auch bei der akuten Serverkonsolidierung einsetzbar.

Verfahren 3 besitzt das Ziel einer kurzen Migrationsdauer bei geringer Dienstunterbrechung und bietet sich folglich für akute Serverkonsolidierungen an. Dazu überträgt es das Schema-Grundgerüst an den Zielserver und verfolgt bei der gewählten Paketgröße und -wartezeit das Ziel einer schnellen Übertragung der Mandantendaten. Durch die Parameterwerte kann die Migrationsdauer im Vergleich zu den Verfahren 1 und 2 weiter reduziert werden. *Abbildung 4b* zeigt jedoch die hohe Mehrbelastung während der Migration aufgrund der langen Sperren bei der Deserialisierung. Der geringe Transaktionsdurchsatz hat zur Folge, dass weniger als 1 % der Seiten durch den Zielserver angefragt werden. Durch die Pufferung von Seiten im Rahmen der Anforderungsphase erreicht das Verfahren bereits wenige Sekunden nach Abschluss der Anforderungsphase die maximale Performance.

Verfahren 4 simuliert das in Kapitel 2 aufgegriffene Verfahren Stop-And-Copy. Nach der Bereitstellung des Schema-Grundgerüsts sorgt die gewählte Paketgröße von 2 GB für die Anforderung sämtlicher Mandantendaten im Rahmen eines einzigen Datenpakets. Dessen Anforderung verhindert eine parallele Transaktionsverarbeitung auf dem Zielserver und verursacht eine 18-sekündige Ausfallzeit. Mangels Übertragung des Pufferstatus hat dieses Verfahren zudem eine Mehrbelastung auf dem Zielserver zur Folge (siehe *Abbildung 4b*). Die maximale Performance für den Mandanten wird in der Folge erst 35 Sekunden nach Abschluss der Anforderungsphase erreicht. Das Verfahren sollte nur bei der Migration inaktiver Mandanten in Betracht gezogen werden.

Die Synchronisationsphase kann bei allen Verfahren in maximal 30 Millisekunden abgeschlossen werden (siehe Abbildung 5). Dies ist insbesondere auf die geringe Transaktionsgröße von 10 Anweisungen zurückzuführen, wodurch der Proxy-Server die Verbindungen schon nach einigen Millisekunden Verarbeitungszeit auf den Zielserver umleiten kann. Entsprechend werden nur wenige Transaktionen parallel ausgeführt, kaum Seiten durch den Zielserver angefordert und keine Seite zurückgefordert. Untersuchungen mit größeren Transaktionen zeigten, dass mehrfaches Übertragen von Seiten bei `DUPLEX_ALL` und `DUPLEX_UNMODIFIED` den Kommunikationsaufwand nur geringfügig erhöht, jedoch zu deutlich weniger Transaktionsabbrüchen führt. Die Ergebnisse weiterer Untersuchungen sind in [Göb15] zu finden.

5 Verwandte Arbeiten

In diesem Kapitel betrachten wir existierende Ansätze zur Live-Migration für mandantenfähige Datenbanksysteme sowie Konzepte in anderen Forschungsgebieten mit vergleichbarem Anforderungsprofil. Das Alleinstellungsmerkmal KitMigs liegt in der Unterstützung verschiedener Migrationsabläufe. Zudem ist es das erste Verfahren für Shared-Nothing-Architekturen, welches auf *Shared Process* basiert. Die existierenden Verfahren umgehen die Herausforderungen gemeinsam genutzter Datenbanken durch das Migrieren kompletter Datenbanken oder Datenbanksysteme.

Elmore et al. [EDAEA11] veröffentlichten mit *Zephyr* ein Verfahren zur Live-Migration in Shared-Nothing-Datenbanksystemen. Der Verlauf des Verfahrens entspricht der Kombination der KitMig-Module `PUSH_WIREFRAME`, `SIMPLEX` und `PULL_ONLY`. Als reaktives Verfahren eignet es sich insbesondere für Serverentlastungen, ist jedoch für die akute Entlastung aufgrund des Verzichts auf die Pufferübertragung nicht empfehlenswert. Da sowohl KitMig als auch Zephyr in das DBMS H2 integriert wurden, haben sie viele Design-Entscheidungen und Konzepte gemein. Zephyr verfolgt das Ziel, Komplexität zu vermeiden, indem es beispielsweise mehrfache Seitenübertragungen strikt verbietet und keine Informationen zu Pufferinhalten an den Zielserver überträgt. Bei diesen Einschränkungen und den resultierenden Schwächen Zephyrs setzt KitMig durch die Bereitstellung diverser Module und Parameter an.

Slacker [BCM+12] ist ein proaktives Verfahren zur Migration kompletter Datenbankinstanzen. Initial erstellt es eine Datenbanksicherung, überträgt sie zum Zielsystem und spielt sie dort ein. Anschließend wird der Zielserver iterativ über Datenänderungen durch verarbeitete Transaktionen informiert. Sind nur noch wenige Änderungen zu übertragen, wird die Transaktionsverarbeitung der Datenbankinstanz gestoppt und ohne Synchronisation auf das Zielsystem umgestellt. Streng genommen führt es somit keine Live-Migration durch. Analog zu KitMig bietet Slacker eine dynamische Drosselung der Seitenübertragungsgeschwindigkeit zur Reduzierung von Mehrbelastungen. Aufgrund seiner proaktiven Vorgehensweise und seines Verzichts auf das Übertragen von Pufferinhalten ist es lediglich für die vorausplanende Serverkonsolidierung geeignet. Als hybrides Verfahren bietet KitMig kein vergleichbares Verfahren.

Das Migrationsverfahren *Albatross* [DNAA10, DNAEA11] wurde für Shared-Disk-Datenbanksysteme entwickelt. Aufgrund des gemeinsamen Speichersystems migriert es keine persistenten Daten, sondern konzentriert sich auf das Verschieben des Pufferinhalts eines Mandanten. Analog zu Slacker überträgt es zu Beginn den kompletten Pufferinhalt und anschließend iterativ dessen Änderungen an den Zielserver. Bei der Lastmigration wird im Gegensatz zu Slacker der Status aktiver Transaktionen migriert, wodurch es sich auch grundsätzlich von allen KitMig-Verfahren unterscheidet, bei denen Transaktionen stets durch einen Server verarbeitet werden. Albatross eignet sich für vorausplanende Serverentlastungen und -konsolidierungen.

ProRea [SCM13] ist das erste veröffentlichte hybride Verfahren zur Live-Migration in Shared-Nothing-Datenbanksystemen und ist vergleichbar mit einem KitMig-Verfahren, bestehend aus `PUSH_BUFFER`, `DUPLEX_ALL` und `PULL_ONLY`. Durch eine zweistufige

Bereitstellungsphase und den Einsatz von Snapshot Isolation verzichtet es auf einen derart restriktives Sperrsystem wie KitMig. Dadurch erlaubt es jedoch nicht serialisierbare Ablaufpläne. Beide Verfahren verwalten Eigentumsrechte an Seiten für den Quell- und Zielserver. Die Bereitstellung des kompletten Pufferinhalts eines Mandanten führt gemäß Abschnitt 3.3 bei aktiven Mandanten zu einer späten Lastmigration, weshalb *ProRea* für akute Serverentlastungen ungeeignet und für vorausplanende Entlastungen nur bedingt geeignet ist. Für die Serverkonsolidierung stellt es hingegen einen geeigneten Ansatz dar.

Das Anforderungsprofil der Live-Migration ist in vergleichbarer Form in verschiedenen Forschungsgebieten vorzufinden. Die dabei entwickelten Konzepte und Verfahren können zum Teil auf die Live-Migration in relationalen Datenbanksystemen übertragen werden. Seit geraumer Zeit werden Migrationsverfahren für Betriebssystemprozesse [PM83] untersucht. Unter der Voraussetzung eines gemeinsamen Speichersystems migrieren sie die Arbeitsspeicherinhalte und Prozesszustände an ein Zielsystem. Die Erkenntnisse wurden auf die Live-Migration von VMs übertragen [CFH+05, NLH05]. Für VM-Migrationen zu entfernten Rechnern mittels eines überregionalen Netzwerks (Wide Area Network, WAN) bedarf es der Migration von VMs und persistenter Daten. Bradford et al. [BKFS07] schlugen hierfür ein proaktives Verfahren vor, dessen Ablauf und Drosselungsmöglichkeiten große Ähnlichkeiten zu Slacker aufweisen. Fuzzy Reorganisation [SBC97] zur Reorganisation relationaler Datenbanken im laufenden Betrieb besitzt Herausforderungen, die mit denen der Live-Migration vergleichbar sind. Im laufenden Betrieb wird dabei eine Kopie der Daten erstellt, reorganisiert und unter Berücksichtigung zeitgleich durchgeführter Änderungen durch eine finale Synchronisation in das aktive System integriert.

6 Zusammenfassung und Ausblick

Die Live-Migration von Mandantendaten gehört zu den wesentlichen Herausforderungen mandantenfähiger Datenbanksysteme als Schlüsselkomponente zur Realisierung von Lastverteilung und Elastizität. In diesem Beitrag stellten wir die Migrationslösung KitMig vor, die für die Live-Migration mandantenfähiger Datenbanksysteme mit Shared-Nothing-Architekturen entwickelt wurde. Wir skizzierten die in KitMig verwendeten Phasen und gaben einen Einblick in die Integration des Systems in das Open-Source-DBMS H2. Die Ergebnisse unserer Untersuchungen verdeutlichten, dass eine geeignete Kombination der bereitgestellten Module und Parameterwerte die Anpassung des Migrationsablaufs an verschiedene gegebene Anforderungen ermöglicht.

Der Einsatz von KitMig ist mit Einschränkungen verbunden, deren Aufhebung das Ziel weiterer Arbeiten ist. Die größte Bedeutung kommt dabei der Entwicklung einer logbasierten Systemwiederherstellung zu, um die Dauerhaftigkeit von Transaktionen während der Migration zu gewährleisten. Zudem arbeiten wir an einer iterativen Datenübertragung in der Bereitstellungsphase, um die Dauer der notwendigen Blockade von modifizierenden Transaktionen zu reduzieren. Mögliche Erweiterungen bestehen in der Komprimierung zu übertragender Datenpakete sowie in der Pausierung aktuell nicht ausführbarer Transaktionen unter Bildung einer Warteschlange statt der sofortigen Veranlassung eines Transaktionsabbruchs. Flexible Migrationsverfahren wie KitMig führen zu neuen Herausforderun-

gen für Migrationsstrategien wie der Berücksichtigung der Eigenschaften verschiedener Migrationsabläufe bei der Entscheidungsfindung sowie der Festlegung des geeigneten Migrationsablaufs in Abhängigkeit vom Anforderungsprofil und Systemzustand. Die Herausforderungen zeigen offene Problemstellungen für zukünftige Arbeiten auf.

Literatur

[AEDE11] Divyakant Agrawal, Amr El Abbadi, Sudipto Das und Aaron J. Elmore. Database Scalability, Elasticity, and Autonomy in the Cloud. In *DASFAA*, Seiten 2–15, 2011.

[AJKS09] Stefan Aulbach, Dean Jacobs, Alfons Kemper und Michael Seibold. A Comparison of Flexible Schemas for Software as a Service. In *SIGMOD*, Seiten 881–888, 2009.

[AJPK09] Stefan Aulbach, Dean Jacobs, Jürgen Primsch und Alfons Kemper. Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen. In *BTW*, Seiten 544–555, 2009.

[BCM+12] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacıgümüş und Prashant Shenoy. „Cut Me Some Slack“: Latency-aware Live Migration for Databases. In *EDBT*, Seiten 432–443, 2012.

[BKFS07] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann und Harald Schiöberg. Live Wide-area Migration of Virtual Machines Including Local Persistent State. In *VEE*, Seiten 169–179, 2007.

[BZP+10] Cor-Paul Bezemer, Andy Zaidman, Bart Platzbeecker, Toine Hurkmans und Aad 't Hart. Enabling Multi-Tenancy: An Industrial Experience Report. In *ICSM*, Seiten 1–8, 2010.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt und Andrew Warfield. Live Migration of Virtual Machines. In *NSDI*, Seiten 273–286, 2005.

[CJP+11] Carlo Curino, Evan Jones, Raluca Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan und Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, Seiten 235–240, 2011.

[CJZM10] Carlo Curino, Evan Jones, Yang Zhang und Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan und Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, Seiten 143–154, 2010.

[DNAA10] Sudipto Das, Shoji Nishimura, Divyakant Agrawal und Amr El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Bericht 2010-09, University of California, Santa Barbara, 2010.

[DNAEA11] Sudipto Das, Shoji Nishimura, Divyakant Agrawal und Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.*, 4(8):494–505, 2011.

[EDAA11] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal und Amr El Abbadi. Towards an Elastic and Autonomic Multitenant Database. In *NetDB (SIGMOD-Workshop)*, 2011.

[EDAEA11] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal und Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, Seiten 301–312, 2011.

[EDP+13] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi und Xifeng Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD*, Seiten 517–528, 2013.

[Göb14a] Andreas Göbel. H2 Proxy – Dynamic Load Balancing for Multi-Tenant Database Systems. *it – Information Technology*, 56(3):127–133, 2014.

[Göb14b] Andreas Göbel. MuTeBench: Turning OLTP-Bench into a Multi-Tenancy Database Benchmark Framework. In *CLOUD COMPUTING*, Seiten 84–87, 2014.

[Göb15] Andreas Göbel. Flexible Live-Migration zur dienstgütebasierten Ressourcenverwaltung in mandantenfähigen Datenbanksystemen. Dissertation (in Vorbereitung), Friedrich-Schiller-Universität Jena, 2015.

[JA07] Dean Jacobs und Stefan Aulbach. Ruminations on Multi-Tenant Databases. In *BTW*, Seiten 514–521, 2007.

[LHM+13] Ziyang Liu, Hakan Hacıgümüş, Hyun Jin Moon, Yun Chi und Wang-Pin Hsiung. PMAX: Tenant Placement in Multitenant Databases for Profit Maximization. In *EDBT*, Seiten 442–453, 2013.

[NLH05] Michael Nelson, Beng-Hong Lim und Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *ATEC*, Seiten 391–394, 2005.

[PM83] Michael L. Powell und Barton P. Miller. Process Migration in DEMOS/MP. *SIGOPS Operating Systems Review*, 17(5):110–119, 1983.

[Rah94] Erhard Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley-Verlag, 1994.

[Rei10] Berthold Reinwald. Database Support for Multi-tenant Applications. *WISS (ICDE Workshop)*, 1:2, 2010.

[SBC97] G. H. Sockut, T. A. Beavin und C.-C. Chang. A Method for On-line Reorganization of a Database. *IBM Systems Journal*, 36(3):411–436, 1997.

[SCM13] Oliver Schiller, Nazario Cipriani und Bernhard Mitschang. ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *EDBT*, Seiten 53–64, 2013.

[SJK+13] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin und Dean Jacobs. RTP: Robust Tenant Placement for Elastic In-memory Database Clusters. In *SIGMOD*, Seiten 773–784, 2013.

[SK12] Michael Seibold und Alfons Kemper. Database as a Service. *Datenbank-Spektrum*, 12(1):59–62, 2012.

[WB09] Craig D. Weissman und Steve Bobrowski. The Design of the Force.Com Multitenant Internet Application Development Platform. In *SIGMOD*, Seiten 889–896, 2009.

[YQR+12] Tao Yu, Jie Qiu, Berthold Reinwald, Lei Zhi, Qirong Wang und Ning Wang. Intelligent Database Placement in Cloud Environment. In *ICWS*, Seiten 544–551, 2012.

[YSY09] Fan Yang, Jayavel Shanmugasundaram und Ramana Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In *CIDR*, 2009.

The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management

Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath,
Steffen Friedrich, Norbert Ritter

Database and Information Systems Group
University of Hamburg
{gessert, xschaars, wingerath, friedrich, ritter}@informatik.uni-hamburg.de

Abstract: The expiration-based caching model of the web is generally considered irreconcilable with the dynamic workloads of cloud database services, where expiration dates are not known in advance. In this paper, we present the *Cache Sketch* data structure which makes expiration-based caching of database records feasible with rich tunable consistency guarantees. The Cache Sketch enables database services to leverage the large existing caching infrastructure of content delivery networks, browser caches and web caches to provide low latency and high scalability. The Cache Sketch employs Bloom filters to create compact representations of potentially stale records to transfer the task of cache coherence to clients. Furthermore, it also minimizes the number of invalidations the service has to perform on caches that support them (e.g., CDNs). With different age-control policies the Cache Sketch achieves very high cache hit ratios with arbitrarily low stale read probabilities. We present the *Constrained Adaptive TTL Estimator* to provide cache expiration dates that optimize the performance of the Cache Sketch and invalidations. To quantify the performance gains and to derive workload-optimal Cache Sketch parameters, we introduce the YCSB Monte-Carlo Caching Simulator (*YMCA*), a generic framework for simulating the performance and consistency characteristics of any caching and replication topology. We also provide empirical evidence for the efficiency of the Cache Sketch construction and the real-world latency reductions of database workloads under CDN-caching.

1 Introduction

In today's cloud data management, most database-as-a-service (DBaaS) systems are exposed through REST/HTTP interfaces. REST APIs make it easy for applications to interact with database services and allow service providers and application designers to leverage the mature, well-researched HTTP protocol and infrastructure. This is particularly true for "NoSQL" systems where the central operations create, read, update, delete (CRUD) map well to REST and HTTP semantics. Yet today, to the best of our knowledge, no DBaaS is capable of exploiting the expiration-based HTTP caching model and its rich, globally distributed content-delivery infrastructure. The reason lies in the impossibility to predict the correct expiration date for database records - any unexpected write operation would entail reads from stale cached copies that did not yet expire. The key insight to the Cache Sketch solution presented in this paper is that the task of cache invalidation can be

shifted from the server to the client using an appropriate data structure.

Caching support in database REST APIs is not only critical for applications that are geographically distributed from the physical location of the database service. It is even more important for mobile and web applications whose performance is governed almost exclusively by latency. Large-scale web sites therefore allocate significant resources to manually optimize caching of static content like images, scripts and style sheets. However, since the recent shift to smarter clients and single-page applications, dynamic database content is increasingly requested in end devices directly, either through data APIs of custom application servers or "backend-as-a-service" systems. This makes data requests extremely latency critical, as they block the user experience. Various studies have quantified the dramatic effects of latency on user satisfaction. For instance, Amazon has found that 100ms of additional latency decrease revenue by 1%. Google similarly discovered that 500ms of additional page load time decrease traffic by 20%[1].

To tackle this problem of significant practical relevance, we introduce a DBaaS caching methodology that employs automatic caching of database records requested through a REST/HTTP API. Cache consistency is ensured using a dual approach: *expiration-based* web caches (browser/device caches, forward proxy caches, ISP caches) are kept coherent through client-side revalidations enabled by the Cache Sketch data structure, whereas *invalidation-based* web caches are invalidated by purge requests issued by the database service. The proposed caching methodology is applicable to any data-serving cloud service, but particularly well-suited for database- and backend-as-a-service systems. The *client Cache Sketch* is a Bloom filter of potentially stale records maintained in the database service. To determine whether a record can safely be fetched from caches, clients query the Cache Sketch before reads. If the record's id is contained in the cache sketch, a revalidation request is sent, as intermediate caches might hold a stale copy. The issued HTTP revalidation request instructs caches to check, whether the database record has a different version than the locally cached copy. If a false positive occurs, a harmless revalidation on a non-stale record is performed, which is similar to a cache miss.

Clients leverage the Cache Sketch for three different goals: fast application and session starts (*cached initialization*), cached reads with consistency guarantees (*bounded staleness*) and low-latency transactions (*conflict-avoidant optimistic transactions*). For *cached initialization*, clients transparently store every fetched record in the client cache (usually the browser cache). At the begin of a new session or page visit, the Cache Sketch is transferred, so clients can check which cached copies from the last session are still up-to-date. The number of necessary requests is thus reduced to the cache miss ratio of intermediate caches. To maintain Δ-*bounded staleness*, the Cache Sketch is refreshed in intervals of Δ. The interval constitutes a controllable upper bound on the staleness of loaded records. Similarly, *conflict-avoidant optimistic transactions* load the Cache Sketch at transaction begin. Subsequent transactional reads exploit cached records, reducing the overall duration and associated abort probability of the transaction.

By optimistically caching all records and employing the Cache Sketch to only revalidate stale records, the same cache hit ratio is achieved as if the time to the next write was known

[1] see http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx (12/12/2014) for both claims

in advance. A record can only be removed from the Cache Sketch once it is sure to have been expired in all caches. Thus, precise estimations of expiration times impact cache hit ratios after writes as well as the number of necessary invalidations. To tune the inherent trade-off between cache hits, stale reads, invalidations and false positives towards a given preference, we present the *Constrained Adaptive TTL Estimator* (CATE) that complements the Cache Sketch by adjusting cache expiration to optimize the trade-off.

The contributions of this paper are threefold:

- We propose the **Cache Sketch** as a data structure to enable the use of expiration- and invalidation-based web caching for cloud data management systems to combine the latency benefits of caching with rich consistency guarantees.
- We describe the Constrained Adapative TTL Estimation (**CATE**) algorithm that computes record expiration dates to minimize stale read probabilities and invalidations while maximizing cache hits.
- We present the Monte-Carlo caching simulation Framework **YMCA** that allows to analyze and optimize caching strategies and Cache Sketch parameters for pluggable network, database and caching topologies.

The paper is structured as follows. Section 2 presents the Cache Sketch and its properties and effects. Section 3 outlines the TTL estimation problem and a possible solution. Section 4 introduces the YMCA simulation frameworks and presents simulated and real empirical results for the proposed combination of web caching and cloud data management. Sections 5 examines related work and Section 6 concludes.

2 Staleness Avoidance through Cache Sketches

The expiration-based caching model of HTTP was deliberately designed for scalability and simplicity. It therefore lacks cache coherence protocols and assumes a static TTL (time to live) indicating the time span for which a resource is valid, allowing every cache to keep a copy. This model works well for immutable content, for example a particular version of JavaScript library. With the rise of REST APIs for cloud services however, this model fails in its naive form - TTLs of dynamic content, in particular database records and query results are not known in advance. This has lead to database interfaces that forbid caching in the first place as otherwise staleness would be uncontrollable.

Figure 1 shows an architectural overview of how our Cache Sketch approach addresses this problem. Every cache in the request path serves cached database records requested by their key to the client, which can either be an end-user's browser, a mobile application or an application server. The Bloom filter of the client Cache Sketch is queried to send a request either as normal request (record not contained) or a revalidation (record contained). The revalidation forces caches to update their copy using an HTTP request conditioned over the record's version (*Etag*).

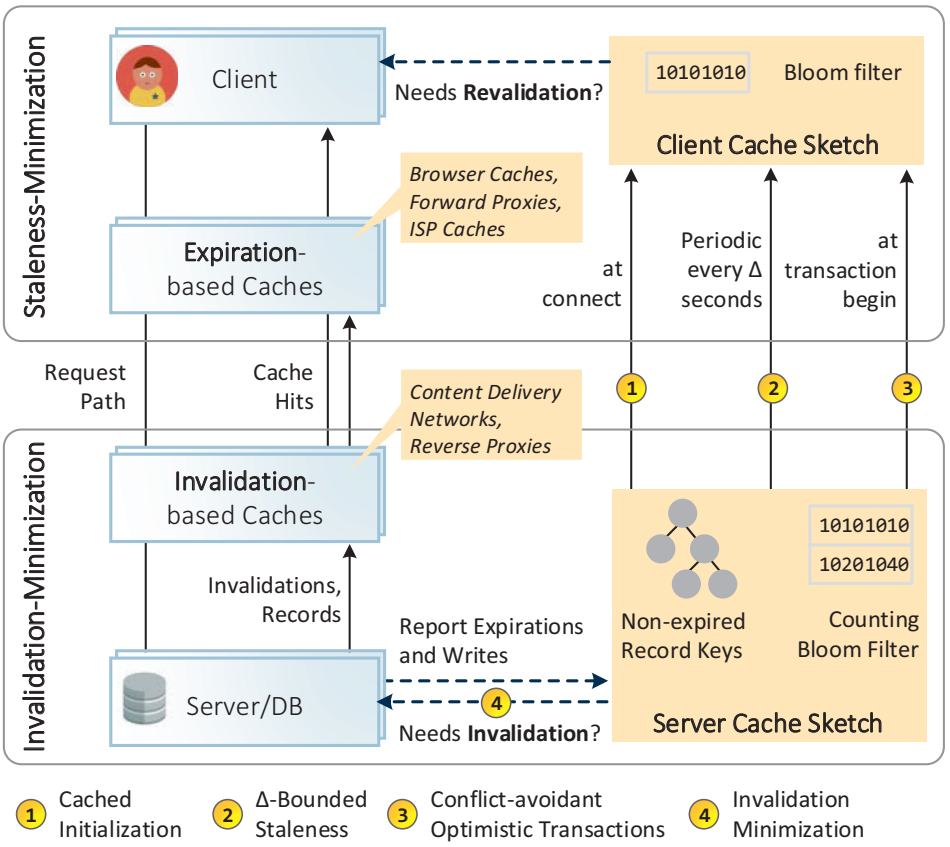


Figure 1: Architectural overview of the client and server Cache Sketch.

The database service tracks the highest TTLs provided at a cache miss. On a subsequent write, the record is added to the Counting Bloom filter of the server Cache Sketch and removed when the record is expired. The database service is furthermore responsible for purging records from invalidation-based caches (CDNs and reverse proxy caches), which allows them to answer revalidations. To minimize invalidation broadcasts, purges are only sent, if the server Cache Sketch reports a record as non-expired. It is important to note, that this scheme does not require any modifications of the HTTP protocol or web cache behavior. The proposed Cache Sketch approach is not specific to a particular database service architecture and can be realized either directly in the nodes of database system or as a tier of stateless REST servers exposing the database. We chose the latter approach, building on the database-independent ORESTES middleware [GBR14].

There are several advantages of caching database records close to clients. First, cache hits have lower latency and higher throughput than uncached requests, as TCP throughput is inversely proportional to the round-trip time [Gri13]. Second, the database service is

under lower load, as it only has to handle write requests and cache misses. Third, clients profit from requests of other clients, as all caches except the browser/device cache are shared. Fourth, flash crowds, i.e. load spikes caused by unexpected and sudden popularity, are mitigated by caching and do not bring down the database service [FFM04]. Fifth, temporary unavailability of the database service can be hidden for reads by letting CDNs and reverse proxies serve cached records while the service is unreachable.

2.1 The Client Cache Sketch

For each potentially non-expired record x, the client Cache Sketch has to contain its key key_x. For now, we will only consider key/id lookups - the most common access pattern in key-value, document and wide-column stores - and discuss how the scheme can be extended to query results, later. As shown in Figure 2, a read on a key is performed by querying the Bloom filter b_t of the client Cache Sketch c_t^c that was generated at time t. The key is hashed using k independent uniformly distributed hash functions that map from the key domain to $[1, m]$, where m is the bit array size of b_t. If all bits $h_1(key), \ldots, h_k(key)$ equal 1, the record is contained and has to be considered stale.

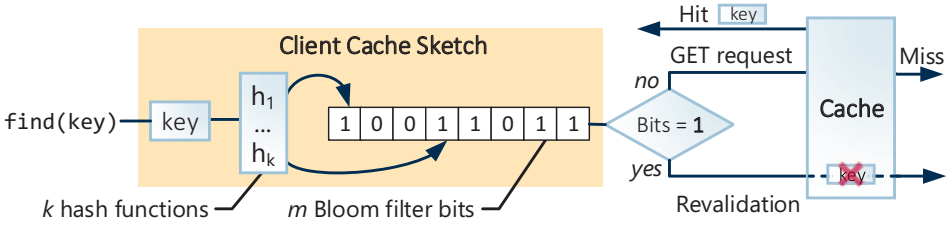


Figure 2: Database read using the Cache Sketch.

Theorem 1 deducts the central guarantee offered by the Cache Sketch using the time-based consistency property Δ-atomicity [GLS11]. Δ-atomic semantics assert that every value becomes visible during the first Δ time units after the acknowledgement of its write.

Theorem 1. *Let c_t^c be the client Cache Sketch generated at time t, containing the key key_x of every record x that was written before it expired in all caches, i.e. every x for which holds that $\exists\, r(x, t_r, TTL), w(x, t_w) : t_r + TTL > t > t_w > t_r$ where $r(x, t_r, TTL)$ is a read (cache miss) on x at time t_r, TTL is the TTL provided for that read and $w(x, t_w)$ is a write on x at time t_w.*

A read on record x performed at time t_x using c_t^c satisfies Δ-atomicity with $\Delta = t_x - t$, i.e. the read is guaranteed to see only records that are at most $\Delta = t_x - t$ time units stale.

Proof. Consider there was a read issued at time t_x using c_t^c that returned a record x that was stale for $\Delta > t_x - t$. This implies that x must have been written at a time $t_w < t$ as otherwise $\Delta < t_x - t$. Now, if there has been no previous read $r(x, t_r, TTL)$ with

$t_r + TTL > t_x$ then x could not have been stale for at least $t_x - t$ time units. If there has been such a read, by the construction of c_t^c the record's key would have been contained in it and the read (a revalidation) could not have been stale. □

Theorem 1 states, that clients can tune the desired degree of consistency by controlling the age Δ of the Cache Sketch: the age directly defines Δ-atomicity. This guarantee relies on the linearizability of the underlying database system, i.e. writes are assumed to be directly visible to uncached reads. If the database is only eventually consistent with Δ_{db}-atomicity, the guarantee is weakened to $(\Delta + \Delta_{db})$-atomicity[2]. Similarly, if the invalidation-based caches only support asynchronous invalidations (which is typical for real-world CDNs [PB08]) with Δ_c-atomicity, the consistency guarantee becomes $(\Delta + \Delta_c)$-atomicity[3]. If Δ_c is an undesired source of uncertainty, Δ-atomicity can be established in two ways. First, invalidation-based caches can be treated as pure expiration-based caches by not letting them answer revalidation requests. The trade-off is that this increases read latency and the load on the database service. Second, invalidations can be performed synchronously. This is a good option for reverse-proxy caches located in the network of the database service. Here, the trade-off is that cache misses have higher latency and can be blocked by the unavailability of a cache node (no partition tolerance).

Definition 1. *A client follows **cached initialization**, if all initial reads are performed using the freshly loaded Cache Sketch c_t^c. A read at t_{now} follows **Δ-bounded staleness**, if it only uses c_t^c if $t_{now} < t + \Delta$. A **conflict-avoidant optimistic transaction** started at t_s uses $c_{t_s}^c$ for transactional reads.*

Definition 1 introduces three client-driven age-control techniques for the Cache Sketch. *Cached initialization* builds on the insight, that initially $\Delta = 0$ for a Cache Sketch piggybacked upon connection. This implies, that every cached record can be used without degrading consistency, i.e. loading the Cache Sketch is equivalent to loading all initially required records in bulk, which may also include all static resources (images, scripts, etc.) of the application or website.
Δ-*Bounded staleness* guarantees Δ-atomicity by not letting the age of the Cache Sketch exceed Δ. Updates may be performed *eagerly* or *lazily*. With eager updates, the client updates c_t^c in intervals of Δ. As this may incur updates despite the absence of an actual workload, lazy updates only fetch a new Cache Sketch on demand. To this end, if a read request is issued at $t_{now} > t + \Delta$, the request is turned into a revalidation instructing the service to append the Cache Sketch to the result. Hence, at the mild cost of a cache miss every Δ time units, c_t^c is updated without additional requests.
Similar to cached initialization, a *conflict-avoidant optimistic transaction* (COT) is started by loading the Cache Sketch. The caching model is only compatible with optimistic transactions as reads are performed in caches which cannot participate in a lock-based concurrency control scheme. By having clients collect the read-sets of their transactions consisting of record ids and version numbers, the database service can realize the transaction

[2] Bailis et al. [BVF+12] have extensively studied the staleness of Dynamo-style systems. They found, that with high probability Δ_{db} is very low and for many configurations not perceivable at all.

[3] We are not aware of any scientific studies on CDN purge latencies. Anecdotally, the Fastly CDN used in our evaluations employs the bimodal multicast protocol for invalidations with measured latencies typically much lower than 200ms: fastly.com/blog/building-fast-and-reliable-purging-system

validation using BOCC+, as shown in [GBR14]. The important alteration that COTs bring to this scheme is that cached reads can drastically reduce the duration T of the transaction, while the Cache Sketch limits staleness to T. Since the abort probability of optimistic transactions has been shown to grow exponentially with T [Tho98], lowering T through cache hits can greatly reduce abort rates[4].

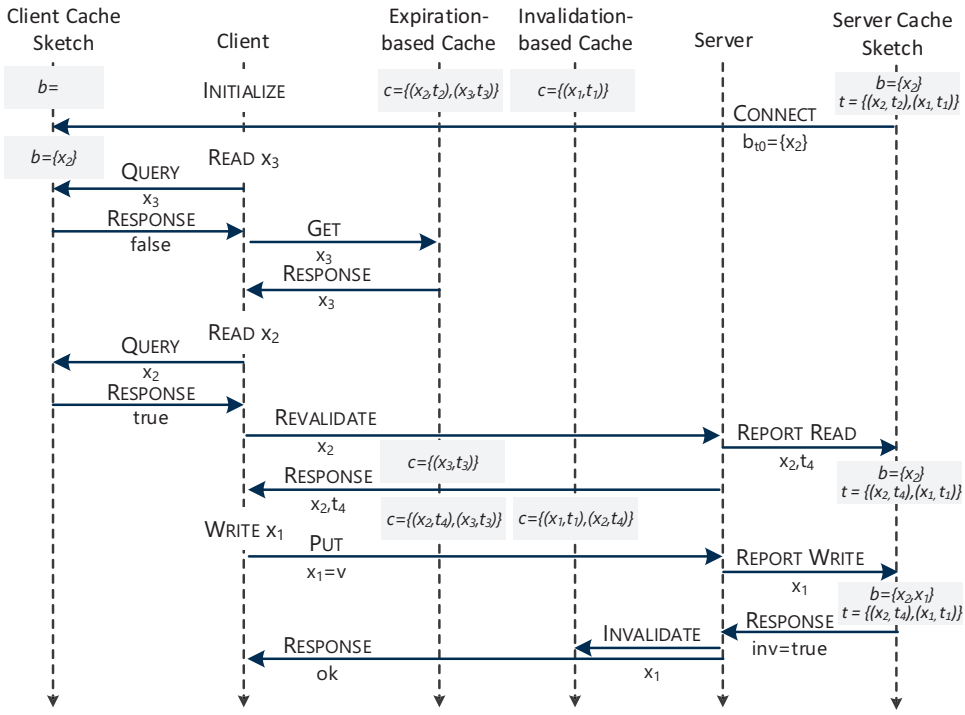


Figure 3: An end-to-end example of the proposed Cache Sketch methodology.

Figure 3 shows an end-to-end example of Cache Sketch usage. First, the client fetches the Cache Sketch. As x_3 is not contained in it, the record is fresh and hence requested normally, resulting in a cache hit. The next record x_2 is contained and hence a revalidation is sent, causing the expiration-based cache to evict its cached copy. The server returns x_2 with a new TTL/expiration date t_4, which is saved in both caches. Additionally, the new expiration date is also reported to the server Cache Sketch, where expiration state is tracked. On the subsequent write on x_1, the server Cache Sketch adds x_1 to the Bloom filter, since its expiration date t_1 has not yet passed. This also tells the server, that invalidation-based caches need to be purged. Any later readers are therefore able to revalidate x_1 from an invalidation-based cache.

[4]We skip many details here. An extensive quantitative investigation is an important part of our future work.

2.2 The Server Cache Sketch

The purpose of the server Cache Sketch c_t^s is the efficient and correct generation of client Cache Sketch defined in Theorem 1. This requires two important capabilities the client Cache Sketch lacks. First, the server Cache Sketch must support *removal* of keys in order to evict expired items. Second, it must support *invalidation queries* which report, whether a write has to be propagated as an invalidation.

Definition 2. *The server Cache Sketch c_t^s consists of a Counting Bloom filter cb_t containing all elements of c_t^c and a mapping of keys to their maximum expiration date $e = \{(k_i, t_i) | max_{t_i = t_r + TTL}(r(x, t_r, TTL) \wedge t_i > t_{now})\}$. When x is updated or deleted, k_x is added to cb_t iff $k_x \in e$. Similarly, an invalidation is only necessary, if $k_x \in e$.*

The employed Counting Bloom filter [BMM02] is an extension of the Bloom filter that allows removals and can be implemented to materialize the corresponding normal Bloom filter, so retrievals of c_t^c do not require any computation. To make the retrieval of the Cache Sketch efficient, the size m of the Bloom filter must be chosen carefully. The false positive rate p is determined by the size m of the bit vector, the number of inserted elements n and the number of hash functions k: $p \approx (1 - exp(-kn/m))^k$. The optimal number of hash functions is $k = \lceil ln(2) \cdot (n/m) \rceil$, giving the size as $m = -n \cdot ln(p)/ln(2)^2$.

A simple model is to choose m so that transferring c_t^c only requires a single round-trip, even at connection startup. This is achieved, if the message size of m bits (and some HTTP metadata) measured in TCP segments of 1,460 bytes does no exceed the initial TCP congestion window size 10, i.e. $m \approx 10 \cdot 1460\, byte = 116800\, bit$. For a false positive rate $p \leq 0.05$, the filter could hence contain up to $n \approx 18732$ distinct records. If n increased to 50000, p would grow logarithmically to $p = 0.326$. If the Bloom filter is only transferred over an already established connection (e.g., after loading an HTML page), it can be significantly larger without incurring an additional round-trip[5].

The server Cache Sketch represents shared state between all server nodes. It lies in the critical request path as read, update and delete operations require modifying it. Previously, we assumed a single c_t^s of every tenant's database. As a generalization, c_t^s can be partitioned and replicated based on tables (resp. buckets, collections, classes) by maintaining a separate c_t^s for each table. This solves two problems. First, updates to the Cache Sketch scale horizontally, mitigating potential write bottlenecks. Second, if an aggregate Cache Sketch for all tables is too large, clients can opt to fetch the Cache Sketch only for the required tables. To expose the aggregate Cache Sketch, the database service assembles the Cache Sketch by performing a union over the respective Bloom filters, which is a simple bitwise OR over their respective bit vectors [BMM02].

To extend the Cache Sketch approach from cached records to cached query results, each cacheable query q has to be identified by a key k_q (e.g., a concatenation of the query and its parameters), so it can be tracked in c_t^s. The database service then evaluates for each record update, whether a matching query q exists and treats it like an update to q itself[6]. An

[5] This is an effect of the TCP slow-start algorithm which continuously increases the congestion window.

[6] From an implementation perspective, this could for instance be achieved through distributed real-time processing frameworks (e.g., Storm, S4, Samza) or well-known techniques for materialized view maintenance.

alternative approach to query caching is to represent query results as lists of record keys, which can then individually profit from caching. The trade-off between both approaches are lower latency of cached results opposed to higher overall hit rates and reuse between queries. Combining both approaches in a single method is part of our future work.

2.3 Quantifying (Δ, p)-Atomicity for the Web Caching Model

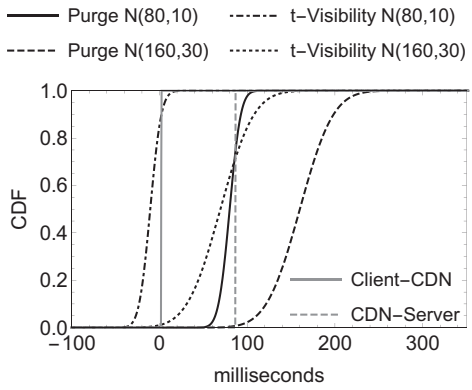
For consistent databases, the Cache Sketch guarantees $(\Delta+\Delta_c)$-atomicity, where Δ_c is the upper bound for the staleness of records read from invalidation-based caches. Δ_c largely overestimates staleness, since access is often local to geographic regions and seldomly governed by worst-case delays. We therefore refine Δ_c to (Δ_c, p)-atomicity[7], which a read satisfies if it is Δ_c-atomic with probability p [BVF+12]. The probability p for (Δ_c, p)-atomic semantics can be expressed through the round-trip latencies T_{cc} (client-cache), T_{sc} (server-cache) and T_i (invalidation). A revalidation or cache miss hitting an invalidation-based cache is Δ_c-atomic, if the time for the corresponding write acknowledgement to travel back to the sender plus the time for the read to reach the cache subtracted from the invalidation latency is smaller than Δ_c:

$$p = Pr[T_i - (T_{sc}/2 + T_{cc}/2 + T_{cc}/2) \leq \Delta_c] \tag{1}$$

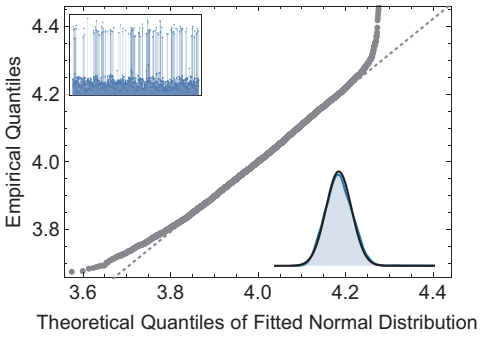
We gathered real-world latency traces to quantify (Δ_c, p)-atomicity and to feed our later simulations with realistic assumptions. The setup consists of a client located in the Amazon EC2 California region, a server in EC2 Ireland and the Fastly CDN as an example of an invalidation-based web caching system. We derived maximum-likelihood distribution fits for T_{cc} and T_{sc} for different continuous distribution families as shown in Figure 4b and 4d, after applying the Tukey-outlier criterion to account for measurement noise, such as the the noisy-neighbor problem. Though there is consensus in the networking literature that in the general case, network delays cannot be modeled using a single distribution [VM06], the normal and Gamma distribution yield good fits for the described setup (goodness-of-fit p-values 0.21 and 0.68 with the Cramér-von Mises test). This is illustrated in the QQ-plot in Figure 4c, which shows that apart from the tails of the raw data (with outliers), the normal distribution describes T_{cc} very accurately.

Based on this data, (Δ_c, p)-atomicity can be computed as shown in Figure 4a with $T_{cc}/2 \sim N(2.00, 0.06)$ and $T_{sc}/2 \sim N(86.54, 0.06)$ for two T_i distributions. For $T_i \sim N(80, 10)$, which we found to be a good upper bound in our experiments, the probability of reading a fresh value starts high and quickly converges to 1. For caches located nearer to the server, p would converge even faster. In conclusion, with asynchronous invalidations exhibiting (Δ_c, p)-atomicity, the Cache Sketch guarantees $(\Delta+\Delta_c, p)$-atomicity. This allows precise reasoning about the consistency trade-off for a given scenario of latency distributions and eases the decision on whether invalidations should be allowed to be asynchronous.

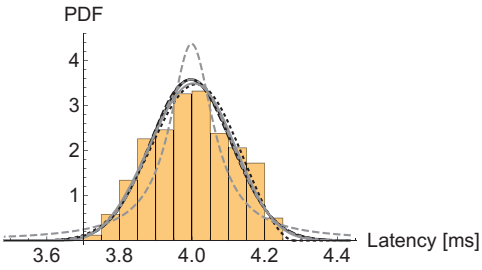
[7] (Δ, p)-atomicity is also referred to as t-Visibility.



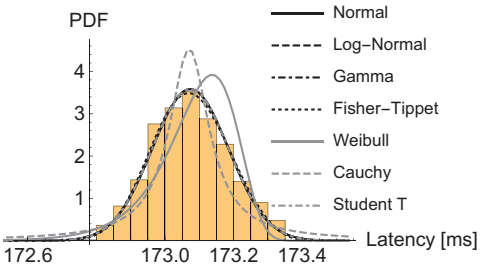
(a) (Δ_c, p)-atomicity in the web caching model.



(c) QQ-Plot for the normal fit of T_{cc}.



(b) Distribution of client-cache latency T_{cc}.



(d) Distribution of server-cache latency T_{sc}

Figure 4: Analysis of exemplary latencies and their effect on (Δ_c, p)-atomicity.

3 Optimal Cache Expiration

The TTL for which caches are allowed to store records significantly affects cache hits, stale reads, invalidations and false positives in the Cache Sketch. For instance, records that experience a write-only workload but are cached with large TTLs would hurt performance, as each write would entail an unnecessary invalidation. Likewise, read-heavy records would suffer from small cache hit ratios, if assigned TTLs are too small. The usefulness of the Cache Sketch depends on its false positive rate. Therefore, we introduce the concept of *TTL estimators* which try to minimize costs.

Definition 3. *A TTL estimator* $E(id, \lambda_m^{id}, \lambda_w^{id}) \rightarrow TTL_{id}$ *maps a record's historic cache miss rate* λ_m^{id} *and write rate* λ_w^{id} *to a TTL that minimizes the cost function:*

$$cost = w_1 \cdot \frac{\#cachemisses}{\#ops} + w_2 \cdot \frac{\#invalidations}{\#ops} + w_3 \cdot \frac{\#stalereads}{\#ops} + w_4 \cdot p$$

The cost function is parameterized by weights w_i that express the relative severity of each condition. For example, in a setup with a slow single server, many invalidation-based

caches and an application with low consistency requirements, w_1 and w_2 would be large to protect the server, while w_3 and w_4 would be smaller.

The estimator is invoked for every cache miss to decide on the next TTL. As a baseline, we propose the *Static Estimator* $E_{static}(id) = TTL_{max}$ that always minimizes cache miss costs through a high static TTL. The trade-off is, that every write on record x happening t seconds before the expiration causes an invalidation, opens the possibility of a stale read caused by the asynchronous invalidation and forces the cache sketch to contain x for the remaining t seconds, increasing its false positive rate. This implies, that the static estimator should only be employed if the Cache Sketch is large enough to hold all records that might be updated in a time window of TTL_{max}. A straight-forward improvement is thus obvious: instead of always estimating very large TTLs, TTLs should rather be correlated to the expected time to the next write on a record. Furthermore, TTLs should also be lower if the workload is write-dominant and higher if it is read-dominant.

To make the improved TTL estimation feasible, some assumptions have to be made. First, we assume, that the per-record workloads are readily available to estimators in the form of cache-miss rates λ_m^{id} and write rates λ_w^{id}. Second, to estimate the probability of writes in certain time intervals, a continuous-time stochastic process of writes $\{W(t), t \in T\}$ is assumed where the random variables $X(t)$ model the amount of writes seen until time t. Intuitively, given that exactly one write happened in the interval $[0, t]$, the time of occurence should be uniformly distributed over $[0, t]$. This requirement is met by the *Poisson process*, which is the most commonly used stochastic process for modeling arrival processes. It is characterized by increments that follow a Poisson distribution $Pr[W(t+s) - W(s) = k] = (\lambda_w t)^k / k! e^{-\lambda_w t}$, where λ_w is the write rate, i.e. the expected amount of writes in a time interval of length t is $E[W(t)] = \lambda_w t$. A very central property for our TTL estimation problem is, that inter-arrival times between writes T_w are exponentially distributed with mean $1/\lambda_w$, i.e. $Pr[T_w < TTL] = 1 - e^{-\lambda_w TTL}$. This implies, that knowing an record's write rate is sufficient information to derive the expected time of the next write $E[T_w] = 1/\lambda_w$ and the quantiles $Q(p, \lambda_w) = -ln(1-p)/\lambda_w$. As the stochastic process of reads is unobservable (hidden through caches), we specifically do neither require knowledge about the *workload mix*, i.e. record-specific read-write ratios nor the *popularity distribution*, i.e. the underlying distribution of record access frequencies. Instead, the TTL estimator implicitly adapts to these conditions.

3.1 Constrained Adaptive TTL Estimation

The goal of the constrained adaptive TTL Estimator (CATE) is to minimize the cost function, while constraining the size of the Cache Sketch to meet a good false positive rate. To this end, CATE adapts TTLs to the cache miss rate λ_r and write rate λ_w instead of merely estimating the time to the next write. The estimation approach is illustrated in Figure 5a: write and cache miss metrics are aggregated in the server and fed into the estimator for each cache miss to retrieve a new TTL. The algorithm is based on four design choices:

1. *Read-only* records yield TTL_{max} and *write-only* records are not cached.

2. If the miss rate λ_m equals the write rate λ_w, the record should be cached for its expected lifetime expressed by the *interarrival time median* of writes $Q(0.5, \lambda_w)$, i.e. the TTL is chosen so that the probability of a write before expiration is 50%.

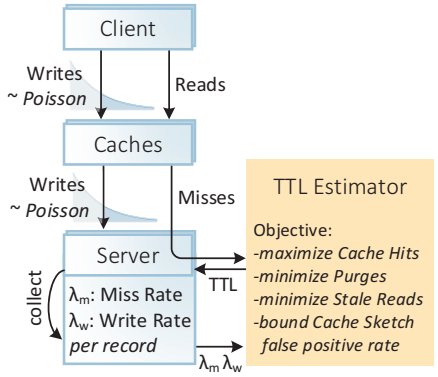
3. A *ratio function* $f\colon \mathbb{R} \to [0,1]$ expresses, how the miss-write ratio impacts the estimated TTLs. It maps the imbalance between misses and writes to p_{target} which gives the TTL as the quantile $Q(p_{target}, \lambda_w)$. If for instance misses dominate writes, $p = 0.9$ would allow a 90% chance of a write before expiration, in order to increase cache hits. Using quantiles over TTLs for the ratio function has two advantages. First, the probability of a write happening before the expiration is easier to interpret than an abstract TTL. Second, the quantile scales with the write rate. The ratio function and its parameters can be tuned to reflect the weights in the cost function.

4. *Constraints* on the false positive rate of the Cache Sketch and the number of invalidations per time period are satisfied by lowering TTLs.

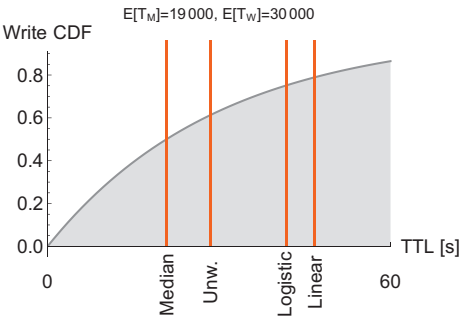
Algorithm 1 Constrained Adaptive TTL Estimation (CATE)

procedure ESTIMATE(λ_m : miss rate, λ_w : write rate) $\rightarrow TTL$
  constants: TTL_{max}, $slope$, f : ratio function
  if $\lambda_w = NIL$ **then return** TTL_{max}
  $$imbalance = \begin{cases} \lambda_m/\lambda_w - 1 & \text{if } \lambda_m \geq \lambda_w \\ -(\lambda_r/\lambda_w - 1) & \text{else} \end{cases}$$
  $p_{max} \leftarrow Pr[T_w < TTL_{max}] = (1 - e^{-\lambda_w TTL_{max}})$
  if f **is** linear **then** $p_{target} \leftarrow 0.5 + slope \cdot imbalance$
  else if f **is** logistic **then** $p_{target} \leftarrow p_{max}/(2p_{max} \cdot e^{-slope \cdot imbalance})$
  else if f **is** unweighted **then** $p_{target} \leftarrow \lambda_m/(\lambda_m + \lambda_w)$
  if Cache Sketch capacity exceeded **then**
    Decrease p_{target} by a penalty proportional to false positive rate
  if Invalidation budget exceeded **then**
    Decrease p_{target}
  $$TTL = \begin{cases} 0 & \text{if } p_{target} \leq 0 \\ TTL_{max} & \text{if } p_{target} \geq p_{max} \\ Q(p_{target}, \lambda_w) & \text{else} \end{cases}$$
  return TTL

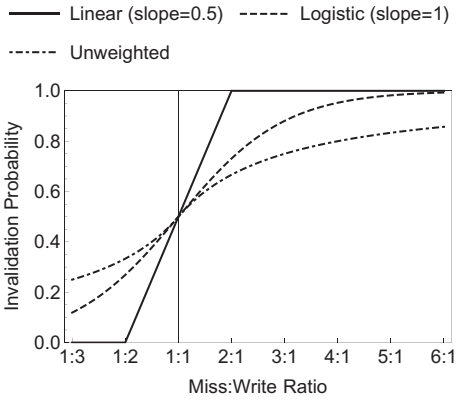
Algorithm 1 describes CATE. The ESTIMATE procedure is invoked for each cache miss. It requires three constants: the maximum TTL TTL_{max}, the ratio function f and the $slope$ which defines how strongly f translates the imbalance between misses and writes into smaller or greater TTLs. First, the miss-write $imbalance$ is calculated. We define it to be 0 if $\lambda_m = \lambda_w$, x if λ_m is x times greater than λ_w and $-x$ if λ_w is x times greater than λ_m. Next, the ratio function maps imbalance to the allowed probability p_{target} of a write (and invalidation) before the expiration date. p_{target} is capped at $p_{max} = Pr[T_w < TTL_{max}]$,



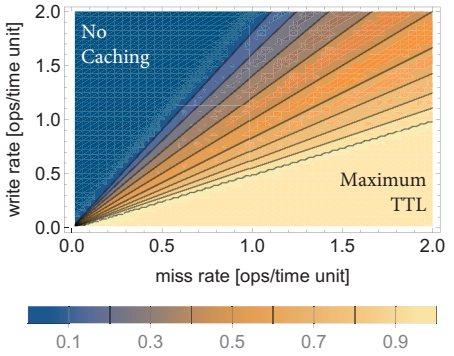
(a) The TTL estimation process.



(b) TTL estimations for an example workload.



(c) Ratio functions.



(d) p_{target} contour plot for linear ratio function.

Figure 5: Constrained Adaptive TTL Estimation.

so that the estimated TTL never gets larger than TTL_{max}. We consider three types of ratio functions shown in Figure 5c: a *linear* and a *logistic* function of the imbalance, as well as the *unweighted* fraction of misses in all operations.

In order not to overfill the Cache Sketch, its current false positive rate is considered. If it exceeds a defined threshold, p_{target} is decreased to trade invalidations on non-expired records against revalidations on expired records. By lowering the probability of writes on non-expired records, Cache Sketch additions decrease, too. Invalidations are treated similarly: if the budget of allowed invalidations is exceeded, p_{target} is decreased. This way, Cache Sketch additions and invalidations are effectively rate-limited. Optimal decrements depend on the severity a of a violation and can be computed as $p_{target} = p_{target} * (1-f)^a$, where f is the degree of violation, for example the difference between the allowed and actual false positive rate. Last, the TTL derived as the quantile $Q(p, \lambda_w)$ is returned.

Figure 5b gives an example of estimated TTLs for a read-heavy scenario, as well as the corresponding probability $Pr[T_w < TTL]$ of a write before expiration. By construction,

all three ratio functions yield a TTL that is higher than the median time between two writes in order to drive cache misses down. The magnitude of this TTL correction is determined by the ratio function and its slope. This makes it obvious, that minimizing the cost function requires tuning of the ratio function in order to meet the relative weights between misses, invalidations, stale reads and false positives. As finding the right TTL_{max} and $slope$ in running system is a cumbersome, manual and error-prone process, we introduce a framework in Section 4 that chooses parameters using Monte-Carlo simulations to find the best solution under a given workload and error function. Figure 5d shows the effect of different miss- and write rates as a contour plot of the linear ratio function. In the upper left area, writes clearly dominate misses, so the estimator opts to not cache the record at all - frequent invalidations would clearly outweigh seldom cache hits. In the bottom right area on the other hand, misses dominate writes, so the record is cached for TTL_{max}. The area in between gradually shifts to higher TTLs (values of p_{target}), with the steepness of the ascent varying with the slope.

As explained above, estimating TTLs requires each database service node to have approximations of write and miss *access rates* for each record. To this end, inter-arrival times are monitored and averaged over a time window using a simple moving average (SMA) or exponentially-weighted moving average (EWMA). The space requirements of the SMA are high, as potentially many arrival times for each record have to be tracked, whereas the EWMA only requires a single value. If the space requirements are still too high, sampling is applied. More specifically, exponentially-biased reservoir sampling is an appropriate stream sampling method that prefers newly streamed values over older ones. The reservoir is a fixed-size stream sample, i.e. in this case a map of record ids to their write and miss moving averages. In the approach of load-balanced middleware service nodes, every server already sees an unbiased sample of operations, whereas in the case that Cache Sketch maintenance is co-located with each partitioned database node, only local records have to be tracked, mitigating the space requirements.

4 Evaluation

We have implemented a Yahoo! Cloud Serving Benchmark (YCSB) wrapper for Monte-Carlo simulation (*YMCA*) for arbitrary caching architectures, which runs completely in memory. YCSB [CST+10] is a widely-adopted standard benchmark for CRUD data stores. As shown in Figure 7, YMCA consists of a client that implements the YCSB interface for basic CRUD operations, an arbitrary number of cache layers and additional modules for collecting metrics, in particular stale reads, cache misses and invalidations. Cache layers are stacked onto each other and can model any caching topology (e.g., a CDN or a reverse proxy). Latencies between layers are drawn from pluggable distributions, assuming symmetric latencies.

Overall, YMCA provides a toolbox to analyze caching behavior of multi-layered database infrastructures. The YMCA client tracks and reports stale reads. A read is considered stale, if there was an acknowledged write with a version that is newer than the version the client read and the timestamp from the begin of the read is newer than the write (i.e., the

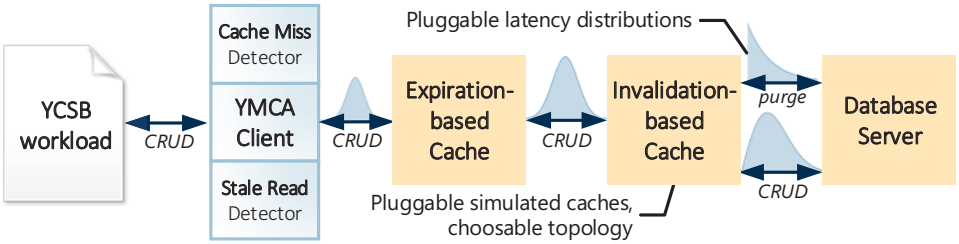


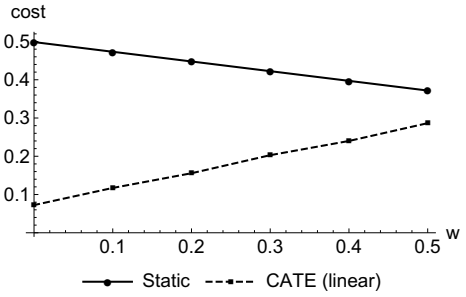
Figure 6: Concept of the extensible YMCA.

read started after the write was acknowledged to the client). Apart from stale reads and invalidations, YMCA also keeps track of cache hits and misses reported by each cache. In order to simulate long durations, YMCA implements a time scaling mechanism: all latencies and TTL estimations can be scaled by a defined factor. In the following, we assume the setting from Section 2.3 that includes an infrastructure consisting of a client, a CDN and the database service. The database service employs the server Cache Sketch to decide, if an update requires an invalidation and passes cache misses to the TTL estimator to assign the record-specific TTL.

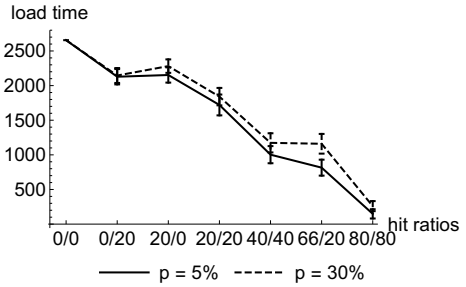
4.1 Parameter Optimization for the TTL Estimators

As discussed above, adaptive TTL estimation depends on the $slope$ of the ratio function, as well as TTL_{max}. In order to optimize these parameters, we use a variation of *maximum descent hill climbing*. Initial slopes of the ratio function are drawn uniformly at random in the $[0, 1]$ range. The algorithm then tests if increasing or decreasing the slope provides an improvement of the simplified cost function $(w \cdot \#cachemisses + (1 - w) \cdot \#invalidations)/\#ops$ that is to be minimized. Since the number of invalidations is an approximate indicator of stale reads as well as a measure of the Bloom filter population, we have found the simplified cost function to be a well-working simplification of the original cost function. Depending on the cost of cache misses compared to invalidations (and the subsumed false positive and stale read rate), terms are weighted with $w \in [0, 1]$.

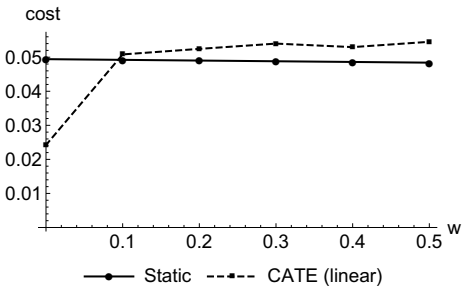
Testing directions of TTL_{max} and $slope$ constitutes a super-step, which concludes by persisting the maximum direction of change (towards a lower cost) for the next super-step to start with. The algorithm terminates after maximum number of super-steps (50) or when it cannot improve costs. Optimizations were performed for YCSB workloads A (write-heavy; read/write balance 50%/50%) and B (read-heavy; read/write balance 95%/5%) with a Zipfian popularity distribution. Each simulation step was run on 100,000 operations for 10 threads, with a targeted throughput of 200 and a time scaling factor of 50, on the default amount of 1000 records. We ran the hill climbing algorithm from 25 starting points. Figure 7a and 7c show the resulting costs as a function of w for the optimized parameters of CATE, with a linear ration function compared to the static TTL estimation



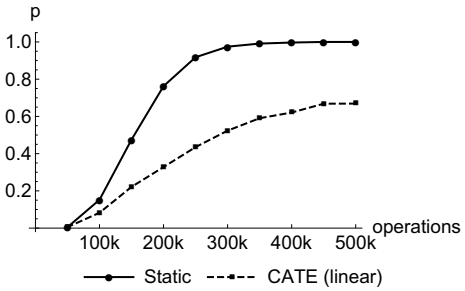
(a) Costs for Workload A.



(b) Page load time improvement through the cached initialization model.



(c) Costs for Workload B.

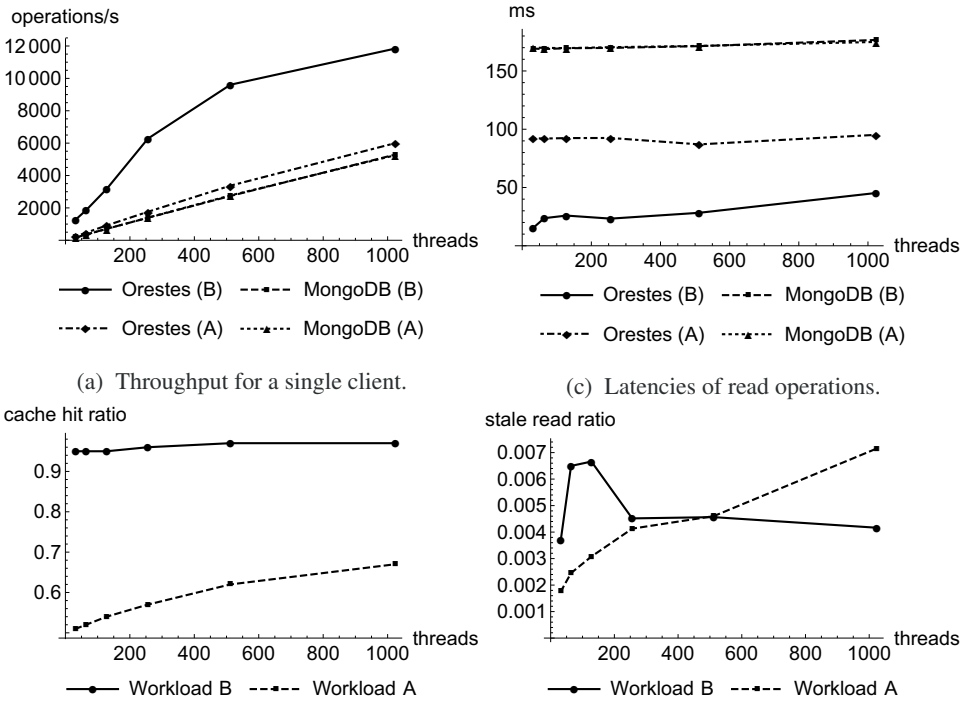


(d) False positive rate for a n=1k, p=0.05 Cache Sketch under Zipf-distributed operations.

Figure 7: YMCA simulation results.

with a high TTL_{max}. The results demonstrate, that CATE performs significantly better than static estimation for applications that do prefer high cache hit rates (workload A). Unsurprisingly, read-heavy workloads leading to many cache hits perform slightly better with a static (maximum) TTL estimation (unless cache misses are weighted very low).

As page load time is arguably the most important web performance metric, we analyzed the gains of *cached initialization* for different browser cache/CDN cache hit rates and two Cache Sketch false positive rates, assuming an average web page with 90 resources using 6 connections [GBR14] and that the Cache Sketch is used for every resource. The results shown in Figure 7b are as drastic as expected: for instance, for the 66%/20% cache hit rate described for Facebook photos [HBvR+13], the speedup is over 320% for $p = 0.05$. The development of the Cache Sketch false positive rate is shown in Figure 7d for 100k records, workload B, a slope optimized for 100k operations and the Bloom filter configured to contain 1k elements with $p = 0.05$. As expected, CATE achieves lower false positive rates by decreasing TTLs, when p grows too large. Even though the Cache Sketch is provisioned to only hold 1/100 of all records, the static estimator performs surprisingly well, as long as the number of operations is smaller than the number of total records.



(a) Throughput for a single client.

(c) Latencies of read operations.

(b) Cache hit ratios.

(d) Stale read ratios.

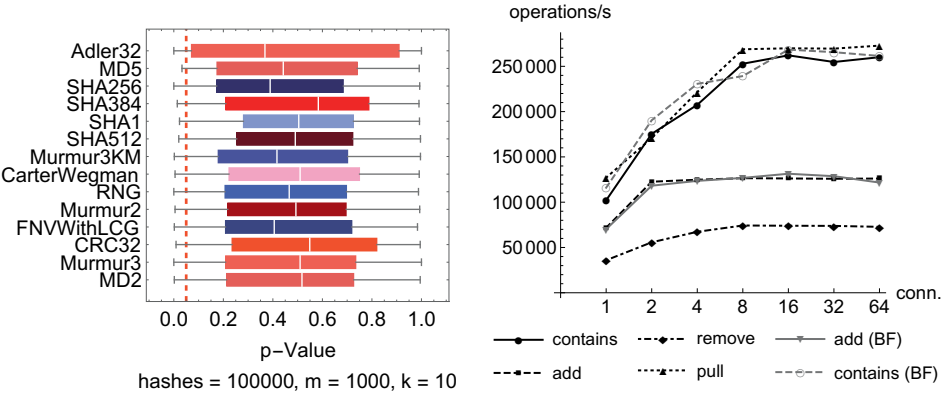
Figure 8: Performance and consistency metrics for YCSB with CDN-caching.

4.2 YCSB Results for CDN-Cached Database Workloads

To validate the results in a real-world setup, we conducted the YCSB benchmark for the described setup on Amazon EC2, using c3.8xlarge instances for the client (*northern California region*) and server (*Ireland*), while caching in the Fastly CDN. We took the document store MongoDB as a baseline and compared it to an ORESTES server running on the MongoDB machine to add the Cache Sketch and the REST API. The benchmark was performed with the same configuration as the simulation, but using the static TTL estimator. Figure 8 shows latency, throughput, cache hit ratios and stale reads for 32 to 1024 threads (i.e. YCSB clients). The results reveal the expected behavior: latency and throughput are considerably improved in both workloads, although a slight non-linearity between 512 and 1024 threads occurs, caused by thread scheduling overhead of the limited single-machine design of YCSB. MongoDB achieves the same latency and throughput in both workloads, since all operations are bound by network latency. The very few stale reads show considerable variance and were largely independent from the number of threads, as seen in Figure 8d, supporting our argument that (Δ_c, p)-atomicity is an appropriate consistency measure and CDNs well-suited to answer Cache Sketch-triggered revalidations.

4.3 Efficient Bloom filter maintenance

The server Cache Sketch requires an efficient underlying Counting Bloom filter. For this purpose, we developed a Bloom filter framework available as an already widely-used open-source project[8]. It supports normal and Counting Bloom filters as in-memory data structures as well as shared filters backed by the in-memory key-value store Redis. The library supports the table-based sharding and replication introduced in Section 3 for high-throughput workloads. The Redis Bloom filter uses the capabilities of Redis to maintain an efficient bit vector for the materialized Bloom filter and relies on pipelining and batch transactions to ensure performance and consistency. The choice of Redis is motivated by its tunable persistence complemented with very low latency.



(a) Quality of Bloom filter hashes for random words (Box-Whisker plot of p-values).

(b) Throughput of the Redis-backed Counting Bloom filter and Bloom Filter (BF).

Figure 9: Analysis of the Redis-backed Bloom filters.

Figure 9 shows selected performance characteristics of the Redis Bloom filters. The uniformity of implemented hash functions for random Strings is evaluated in Figure 9a using the p-values for 100 χ^2-goodness-of-fit tests. For random inputs (e.g., UUID record keys) all hash functions perform reasonably well - including simple checksums. However, for keys exhibiting structure, the best trade-off between speed of computation and uniformity is reached by Murmur 3. Figure 9b plots the throughput of the unpartitioned, non-replicated Redis Bloom filters for a growing amount of connections with $m = 100000$ and $k = 5$ on an Intel quad-core server with 16GB RAM. Read operations (querying, pulling the complete filter) achieve roughly 250k ops/s, while write operations (adding, removing) that require some overhead for counter maintenance and concurrency still achieve over 50k ops/s resp. 100k ops/s. This illustrates, that even a single-server Redis Cache Sketch is not likely to become a bottleneck in a database service.

[8] Available at https://github.com/Baqend/Orestes-Bloomfilter along with much more detailed results.

5 Related Work

Expiration-based web caching of static content has been researched from many perspectives. Huang et al. give an up-to-date analysis of the Facebook photo serving architecture which includes browsers caches, CDNs, custom edge caches and data store-level caching [HBvR+13]. The Summary Cache project [FCAB00] is another example for the use of Bloom filters in caching, where they are employed as metadata digests in cooperative web caches. Candan et al. [CLL+01] pioneered the idea of exploiting invalidation-based web caching for databases with the CachePortal system that detects changes of HTML pages based on underlying SQL queries and triggers corresponding invalidations.

An alternative approach to low latency applications are geo-replicated database systems, where a wealth of new systems and protocols have recently been proposed, including PNUTS, Walter, COPS, Megastore, Spanner, F1 and MDCC [KPF+13]. Some of the earlier approaches like DBCache and DBProxy [APTP03] also relied on caching, however in the form of dedicated database proxies. Geo-replicated approaches explore different positions in the consistency vs. performance trade-off-space, but usually require multiple synchronous wide-area round-trips for a consistency guarantee. Consistency in distributed and replicated storage systems has been studied in both theory [GLS11] and practice: PBS [BVF+12] has a similar approach to YMCA, using Monte-Carlo simulation to determine average staleness of reads in Dynamo-style quorum systems.

Our focus in this paper is different from previous work on web caching and geo-replication, since we aim to enable the use of expiration-based caching for database workloads with tunable Δ-atomicity, relying only on readily available infrastructure and client capabilities.

6 Conclusions

In this paper, we addressed the problem of enabling database services to serve data from the globally-distributed caching infrastructure of the web. The problem is motivated by the observation, that web performance and user-perceived latency are a key differentiator for cloud service and application providers. More specifically, we designed the *Cache Sketch*, a data structure that allows clients to control their desired degree of consistency in the form of Δ-atomicity, while being able to read every non-stale record from expiration-based caches (e.g., browser caches). To this end, the database service maintains the Cache Sketch as a Bloom filter of potentially stale records, while additionally employing it to decide, whether an update operation requires purging of invalidation-based caches (e.g., CDNs). To minimize the Cache Sketch size, invalidation costs and cache misses, we proposed the concept of *TTL Estimators* that produce access-dependent expiration dates. To reason about the resulting performance and consistency, the YCSB Monte-Carlo Caching Simulator offers a generic framework for analyzing different workloads, caching architectures and Cache Sketch parameters. The evaluation of CDN-cached workloads and Counting Bloom filter performance supports our claim, that the Cache Sketch is a feasible approach for large latency reductions in database services.

Our work leaves a number of questions for future investigation. One important area is the combination of the Cache Sketch with optimistic transactions, in particular a quantitative analysis of the abort rate reduction that can be achieved. Another important area is improving the efficiency of the Cache Sketch even further by designing the TTL estimator to learn optimal decisions online, without previous training in simulations, perhaps through techniques of time-series analysis and reinforcement learning.

References

[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the ICDE*, pages 821–831, 2003.

[BMM02] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Math.*, 2002.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[CLL+01] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-driven Web Sites. In *SIGMOD*, pages 532–543, New York, NY, USA, 2001. ACM.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing Content Publication with Coral. In *NSDI*, volume 4, pages 18–18, 2004.

[GBR14] Felix Gessert, Florian Bücklers, and Norbert Ritter. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB 2014*, 2014.

[GLS11] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *ACM PODC*, pages 197–206. ACM, 2011.

[Gri13] Ilya Grigorik. *High performance browser networking*. O'Reilly Media, [S.l.], 2013.

[HBvR+13] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *SOSP*, pages 167–181, 2013.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *EuroSys*, pages 113–126. ACM, 2013.

[PB08] M. Pathan and R. Buyya. A taxonomy of CDNs. *Content delivery networks*, pages 33–77, 2008.

[Tho98] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[VM06] Piet Van Mieghem. *Performance analysis of communications networks and systems*. Cambridge University Press, 2006.

Towards Automated Polyglot Persistence

Michael Schaarschmidt, Felix Gessert, Norbert Ritter

Database and Information Systems Group
University of Hamburg
mks40@cam.ac.uk, {gessert, ritter}@informatik.uni-hamburg.de

Abstract: In this paper, we present an innovative solution for providing automated polyglot persistence based on service level agreements defined over functional and non-functional requirements of database systems. Complex applications require polyglot persistence to deal with a wide range of database related needs. Until now, the overhead and the required know-how to manage multiple database systems prevents many applications from employing efficient polyglot persistence solutions. Instead, developers are often forced to implement one-size-fits-all solutions that do not scale well and cannot easily be upgraded. Therefore, we introduce the concept for a Polyglot Persistence Mediator (PPM), which allows for runtime decisions on routing data to different backends according to schema-based annotations. This enables applications to either employ polyglot persistence right from the beginning or employ new systems at any point with minimal overhead. We have implemented and evaluated the concept of automated polyglot persistence for a REST-based Database-as-a-Service setting. Evaluations were performed on various EC2 setups, showing a scalable write-performance increase of 50-100% for a typical polyglot persistence scenario as well as drastically reduced latencies for reads and queries.

1 Introduction

Polyglot persistence is the concept of using different database systems within a single application domain, addressing different functional and non-functional needs with each system [SF12]. While virtually any non-single-purpose application could benefit from polyglot persistence, there are currently some obvious drawbacks. Designing and implementing an application on multiple databases is considerably harder than just using one backend. Furthermore, application demands begin to exceed capabilities of single databases. At the same time, overhead of configuration, deployment and maintenance increases drastically with each database system used. Therefore today, superior polyglot persistence solutions are often abandoned for lack of know-how and resources.

Recently, cloud providers began to allow programmers to develop and deploy applications at stellar paces. However, using a state-of-the-art cloud services, tenants still have to make the choice of using a certain database [Hac02, Agr09, Cur11]. To solve this dilemma, we are going to present the PPM as a new kind of middleware service layer. Employing a service level agreement (SLA), developers can define specific requirements for their schemas. Schemas are centrally managed by the Orestes [Ges14] middleware, which maps

them to individual database protocols. Even though many NoSQL systems do not employ schemas (although some like DynamoDB or Cassandra do), we assume the presence of a central schema in order to compute a database-independent evaluation of requirements.

For illustration, consider the scenario of an online newspaper: usually, there is a ranking of the most popular articles. To this end, articles have to maintain hit-counters that need to be written up to many thousand times per second. Assume the articles themselves are kept in a document store and the application manages a sorted set of hit-counters for news articles. The article itself is rarely changed after publication. Nevertheless, if hit-counters and articles were stored together, too many writes on the hit-counter would eventually slowdown reads. Using the SLA annotations for write-throughput and another annotation for sorting, the PPM could decide to split objects so that writes on the hit-counter are stored to Redis (achieving much higher throughputs), while reads are directed to MongoDB.

In this paper, we will first introduce a discrete classification of functional and non-functional requirements of database systems, especially NoSQL systems. This enables us to derive the concept of an automated choice of backends. In section 3, we explain the architecture of our prototype. In section 4, we introduce our EC2 benchmark scenario. We will discuss a number of enhancements in section 5.

2 Concept

Automated polyglot persistence requires formal decision criteria. To this end, we first need a classification of functional and non-functional requirements. Table 1 provides an overview of such requirements. On the highest level, requirements are divided into binary and continuous requirements. Binary requirements support yes-or-no decisions: either a database supports server-side joins or it does not. This might be subjective for some non-functional requirements like scalability, since there is no agreed-upon way of measuring it. Continuous requirements like write latency on the other hand can be evaluated by comparing specific values to the context of an application. For instance, low latency for an interactive website usually would not constitute low latency for high-frequency trading. Therefore, automated polyglot persistence needs to provide rich-syntax SLAs that can be parametrized with application-specific goals.

2.1 Requirements

Figure 1 provides an overview of our concept of annotation-based SLAs on those requirements. On a high level, tenants define schemas consisting of databases, classes and fields. Tenants may then define annotations that can be used to annotate complete databases, classes or attributes (fields) of a class. Binary functional and non-functional requirements (capabilities) annotated at a certain hierarchy level result in constraints that have to be met by every entity on and below that level. Continuous requirements are simply pushed down to the field level, i.e., if a complete database is annotated to support 99.5% of read avail-

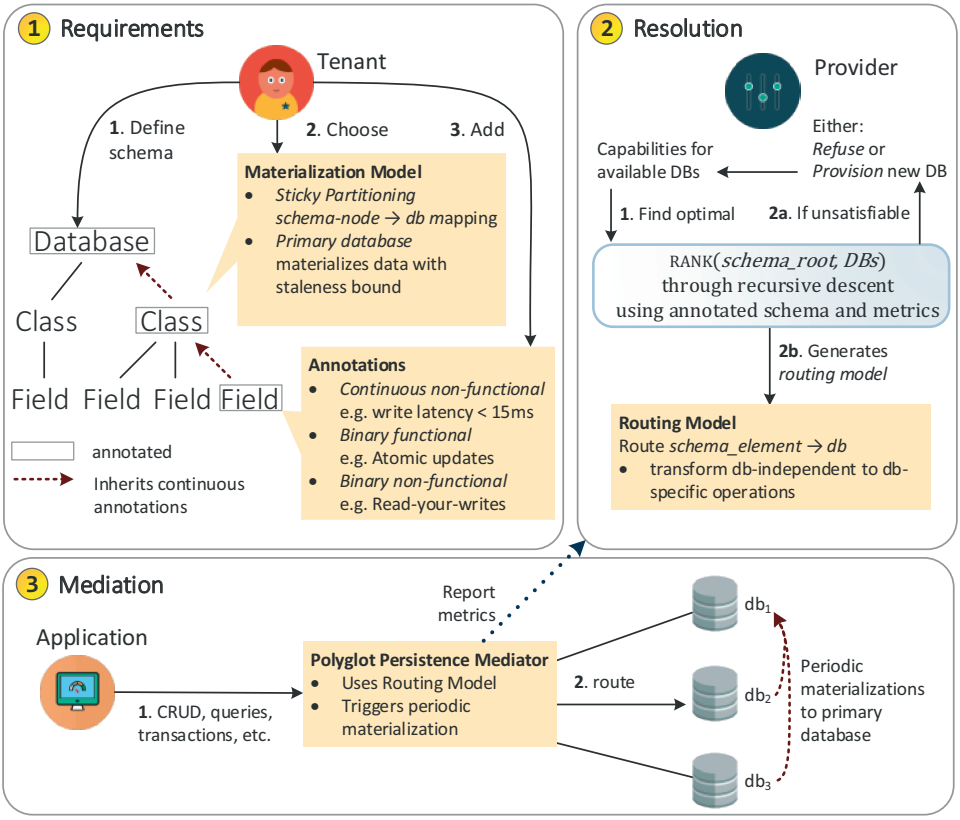


Figure 1: SLA concept and persistence mediation.

ability, every field in that database inherits that annotation. Enforcing continuous requirements on higher levels (i.e., classes) is not feasible, since this would require knowledge about the distribution of this requirement on all its child entities.

An annotation consists of an arbitrary number of binary requirements and continuous non-functional requirements specified either through utility functions or specific goals on the requirement (e.g., latency below 20 milliseconds). In the following section, we are going to demonstrate how these specifications translate to routing decisions by introducing our scoring model. We will also introduce how annotations combined over different hierarchy levels can be resolved. Finally, using polyglot persistence, tenants also have to chose a materialization model: Sticky partitioning instructs the mediator to always route operations for a schema-node to the same database. The primary database defines a read-only master copy to which the mediator periodically materializes data stored in other databases. A staleness bound defines the maximum tolerable delay Δ between materializations. Under the primary copy model, applications must thus tolerate eventual consistency with Δ-atomicity, i.e. the possibility of reading a value that has been stale for at most Δ. This model allows to unite complex, slightly stale queries in the primary database with high

throughput and low latency for simple updates and queries performed in other databases.

2.2 Resolution and scoring

Annotations are resolved by the provider by first comparing the specified binary requirements with all currently available systems. If no system can provide the desired combination, the provider can either reject the annotation right away or try to provision another type of database. In this context, a system is one specific deployment of a database, i.e., an IaaS provider might manage a number of different configurations of the same database. All databases capable of delivering the binary requirements are then scored to find an optimal setup for a specific tenant. The evaluation of annotations is performed recursively over all hierarchy levels of the schema, as shown in algorithm 1.

Annotation	Type	Annotated at
Read Availability	Continuous	*
Write Availability	Continuous	*
Read Latency	Continuous	*
Write Latency	Continuous	*
Write Throughput	Continuous	*
Data Vol. Scalability	Non-Functional	Field/Class/DB
Write Scalability	Non-Functional	Field/Class/DB
Read Scalabilty	Non-Functional	Field/Class/DB
Elasticity	Non-Functional	Field/Class/DB
Durability	Non-Functional	Field/Class/DB
Replicated	Non-Functional	Field/Class/DB
Consistency*	Non-Functional	Field/Class
Scans	Functional	Field
Sorting	Functional	Field
Range Queries	Functional	Field
Point Lookups	Functional	Field
ACID Transactions	Functional	Class/DB
Conditional Updates	Functional	Field
Joins	Functional	Class/DB
Analytics Integration	Functional	Field/Class/DB
Fulltext Search	Functional	Field
Atomic Updates	Functional	Field/Class

Table 1: Proposed annotations.
*Multiple levels of consistency.

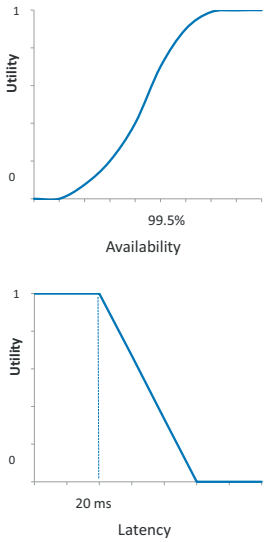


Figure 2: Utility functions

We start at the database (root) level of the schema. First, we exclude all databases incompatible with the current node's constraints (i.e., its binary requirements), so child nodes cannot choose databases their parent nodes do not support. Field nodes simply calculate their scores according to the scoring model below. Other nodes recursively calculate mappings of databases to scores for their child nodes. We then intersect among the resulting databases (line 7) to find systems that can support all requirements of the child nodes and average over the resulting scores of each database of each child node. When the recursion returns to the current node and it is annotated (line 8), it adds the resulting

optimal mapping to the routing model. Finally, (aggregated) scores are returned.

Consider the following example: a class is annotated towards object-level atomic updates. One field of this class requires a certain level of write-throughput, another one has an annotation for read-latency. At the root level, there is nothing to do, so the algorithm turns to class nodes. For the specific class, all databases incapable of ensuring atomicity are removed. Individual field level annotations are now evaluated and each returns a set of databases capable of supporting both their binary and continuous requirements. The recursion then returns and computes the intersection of each field node, determining which databases can best support the required combination of throughput and latency. Finally, the class node adds the result to its routing model and all data items will be stored to the according database. Annotating a node with a binary requirement at database or class level means that all items of this database/class will be stored to the same node. The algorithm ensures compatibility of annotations along the schema hierarchy. It should be noted that schema or annotation changes may require repartitioning, the details of which we leave to future work.

Algorithm 1 Scoring algorithm for input schema node

```
1:  procedure RANK(node, DBs) returns {db → score}
2:      drop db ∈ DB if not node.annotations ⊆ db.capabilities
3:      if node is field then
4:          scores ← {db ∈ DBs → score(db, node)}
5:      else
6:          childScores ← {(child, db, score) |
                child ∈ node.children and
                (db, score) ∈ RANK(child, DBs)}
7:          scores ← db, avg(score) from childScores
                group by db
                having count(child) = |node.children|
8:      if node is annotated then
9:          add (node → argmax_db scores) to routingModel
10:     return scores
```

The score of a database is calculated by adding individual scores for each continuous non-functional requirement $cn \in CN$. Tenants can also assign arbitrary weights to each requirement to model relative importance. The total score is then normalized by the sum of weights.

$$score(db) = \frac{\sum_{i=1}^{|CN|} w_i * f_i(metric(cn_i))}{\sum_{i=1}^{|CN|} w_i} \tag{1}$$

For our scoring model, we propose two alternatives. First, we consider requirement-specific, normalized utility functions. A utility function maps values of a particular metric to the utility this requirement has for a specific use-case: $f(metric) \rightarrow utility \in [0,1]$.

Figure 2 shows two examples of requirement-specific utility. For instance, an interactive application may consider any latency below 20 milliseconds to be acceptable, with a linear

decrease to zero utility at, for instance, 50 milliseconds. On the other hand, availability might be scored by a sigmoid function, indicating that availability below a certain threshold is of zero utility and then drastically increases in utility up to some point of saturation (e.g., 98% vs 99.99% of availability). Practically, users could interactively manipulate these functions in a service dashboard. Normalization of utility functions helps computing a unified score and to define SLA violations. For instance, the SLA may include multiple thresholds with different consequences. First, monitoring such thresholds helps providers to better understand their setups and the impact of changes on its performance. Falling under a certain threshold could also result in auto-scaling the respective database. Second, violations may trigger compensations for tenants in the form of refunds or service credit.

The second scoring model does not require users to specify a mapping of values to utilities. Instead, they simply specify goal values for each requirement. Goals are then compared against current metrics of the system in a manner of performance indexing [LS13]: $f(cn_i) = goal(cn_i)/metric(cn_i)$. For instance, a goal of 50 milliseconds in latency compared to an actual average latency of 20 milliseconds would result in a score of 2.5. Thanks to the collected metrics, arbitrary SLA models may be defined (e.g. pricing models based on deviations [Bas12]). After computing scores, the database with the maximum score will be selected to store the annotated field. This decision is made based on both the current and historic values, using a weighted moving average of all the metrics collected by the provider (i.e., calculating either performance indices or current values of all utility functions). If annotations are changed later, the provider would have to support live data migration between different databases.

2.3 Mediation

The Polyglot Persistence Mediator acts as a broker between applications and backend databases. Applications use a defined interface, e.g., a REST API, to issue queries, CRUD operations, transactions and other operations to the mediator in a database-agnostic fashion. Based on the routing model the mediator selects the appropriate database and transforms the incoming operation to database-specific operations:

$$transform(agnosticOperation, db) \rightarrow dbOperations \tag{2}$$

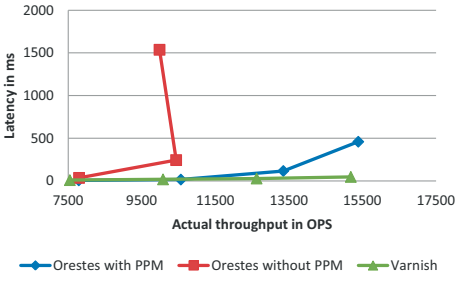
As an example, consider the addition of a value to an array-valued field, for which the resolution step determined MongoDB as most appropriate for the tenant's annotations. The mediator looks up the affected field in the routing model which yields *MongoDB*. Next, the mediator queries its transformation rule repository to map the incoming operation *push(obj, field, val)* to the specific operation {*$push: { field: val }*} and forwards it to the selected MongoDB cluster. In general, an operation can be transformed into a set of operations to account for potential data model transformations and denormalization (e.g., secondary index maintenance).

The mediator is stateless towards clients and can thus be replicated arbitrarily for linear scalability. Updates to the routing model (schema changes) can be shared through coordination services (e.g., Zookeeper [Hun10]), consensus protocols (e.g., Paxos [Lam01], Raft

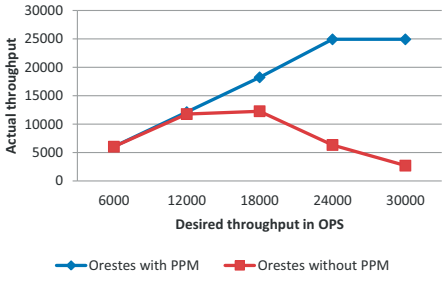
[OO14]) or classic 2PC with different availability/consistency trade-offs [Bak11, Bai12]. The mediator also manages the aforementioned materialization model.

The PPM continuously monitors operations issued to backend databases to report metrics for throughput, latency and availability, aggregating them into means, modes, weighted moving averages, percentiles and standard deviations. The metrics are used for scoring in the resolution step as well as the detection of SLA violations. Scoring thus reflects the actual system state. Of course, when a new database is provisioned, metrics are unknown. They can either be estimated using a system performance model of the database or by running a synthetic workload [Coo10, Sob08] or historic traces. The mediator allows two deployment models: The PPM can be deployed on-premise (e.g., in a private cloud) relying on local database deployments and Database-as-a-Service offers of public cloud providers (e.g. Amazon RDS/DynamoDB/S3, Windows Azure Table Storage).

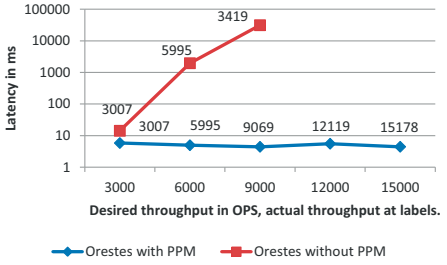
3 Architecture for a Polyglot Persistence Mediator



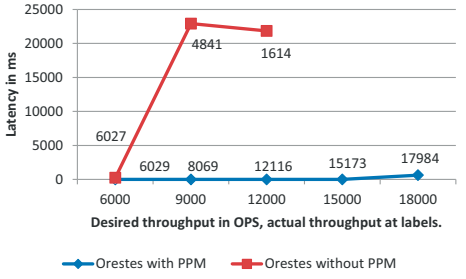
(a) Latency behaviour for a single client/single server set-up.



(c) Write-only benchmark results, 12 clients and 4 servers



(b) Read-only benchmark results, 12 clients and 4 servers.



(d) Mixed benchmark results, 12 clients and 4 servers, 95% reads.

Figure 3: Evaluation of the Polyglot Persistence Mediator.

We implemented a prototype of the PPM as part of a middleware called Orestes, which

provides standardised, REST-based access to different databases [Ges14]. More specifically, applications can use a universal interface which maps all client-side operations to the appropriate server-side protocols, depending on the database an application prefers. Orestes defines a data model consisting of objects and buckets that correspond to, for instance, MongoDB documents and collections. We can thus achieve polyglot persistence on a field level by adding annotations as schema meta data.

In our experiments, we used MongoDB as the principal storage facility (i.e., the primary database materialization model) and Redis as a caching layer to accelerate writes, which constitutes a typical polyglot persistence scenario. The PPM analyses annotations (using simple performance indices) and routes partial updates on objects annotated towards write-throughput, write-latency etc. to Redis. The PPM contains a module that schedules materialisation between databases while providing strong guarantees: strictly ordered materialisation and an upper bound on the materialisation time for any particular object. Similar measures are taken to ensure delete operations are carried out for all databases containing parts of an object.

4 Evaluation

Evaluations were performed using various Amazon EC2 set-ups. To this end, we implemented our own benchmark client, collecting metrics similar to the Yahoo! Cloud Serving Benchmark (YCSB) [Coo10], i.e., averages, min/max and percentiles on latency as well as average throughput.

The first test scenario describes a single client/single server setup for the aforementioned scenario of hit-counters in online newspapers. Access patterns were generated according to a Zipf-distribution. Materialisation intervals were set to 60 seconds. Figure 3a demonstrates typical latency behaviour for medium throughputs. Notably, the actual throughput of Orestes without PPM even decreases on higher loads (backwards curve). MongoDB and Redis each ran on a separate *m1.large* instance, whereas the benchmark client and the Orestes server with/without PPM each ran on a *c3.4xlarge* since the main overhead occurs between client and Orestes server. MongoDB instances were equipped with 1,000 provisioned IOPS for its main storage volume and the write-concern was set to *acknowledged*.

For the benchmark, 100,000 update operations were executed on 100 different articles in MongoDB with hit-counters annotated to be stored in Redis. The line chart shows that the Orestes middle-ware supported by the PPM performs significantly better than Orestes without a PPM. In this context, Orestes without a PPM means that all database operations were routed to a single database (i.e., MongoDB). We found that the throughput-limit corresponds to an average latency of approximately 500 milliseconds. In this setup, the PPM achieved a 50% increase in write-throughput while maintaining lower latencies throughout. For a HTTP baseline, we used Varnish to benchmark performing GET requests against a static resource. At low throughputs, the PPM constantly achieved better latencies than both Varnish and Orestes without a PPM. Subfigures b to d demonstrate results for a set-up of 12 benchmark clients, 4 servers and 1 server for each database (all *m1.large*). Figure 3b

shows results for a read-only benchmark. In our use-case of top-listed articles, we were interested only in the top 10 articles in the sorted set. While articles were stored in MongoDB, PPM-supported reads were routed to the Redis set. Reads without the PPM were simply executed on MongoDB. Visibly, the persistence mediator provides a consistent latency under 10 ms up to 15,000 reads on the sorted set per second. Note that the persistence mediator would still route reads on other fields or the complete object to MongoDB (since we only annotated a specific attribute).

Figure 3c demonstrates the same setup in a write-only scenario. Comparing desired and actual throughputs, we can see that MongoDB can serve up to 12,000 writes per second while the persistence mediator reaches 25,000. Hence, employing the PPM, we used MongoDB as our default storage, but doubled write-throughput for a common scenario.

5 Future Work

Scoring and Database selection. The ability of a cloud provider to maintain its SLA guarantees heavily depends on the scoring. To this end, it is crucial for the provider to select database configurations that indeed fulfill the requirements, i.e., consistently achieve high scores. One line of further research therefore is the estimation of future scores based on historic metrics. The scoring could then be adapted to prefer selections that have expected high scores in the future. There are many potential statistical and machine learning techniques to achieve this. For instance reinforcement learning (e.g., Q-learning [Dut11]) could be used to learn from past selection decisions based on the utility achieved.

Workload Management and Multi-Tenancy. The persistence mediator could improve performance by actively scheduling requests. Requests that pertain to throughput-oriented annotations can be improved by batching while latency-sensitive requests can be scheduled to experience minimum queuing delays. This workload management has to take place at a per-database-level and depends on the multi-tenancy strategy employed by the mediator to ensure sufficient isolation between tenants.

Polyglot Setups. As a practical question, future work needs to consider which system setups might provide optimal polyglot persistence functionality. For our experiments, Redis was used as an on-demand caching layer. Many other combinations are useful, too. For instance, object stores like S3 can be used to store blob data, while shared-nothing file systems such as HDFS could be used for fields with analytic potential, leveraged through platforms such as Hadoop and Spark. Wide-column stores (e.g., HBase) and table stores (e.g., DynamoDB) can similarly be used to store analyzable, structured, write-heavy data.

6 Conclusion

In this paper, we introduced our Polyglot Persistence Mediator approach. It enables tenants to leverage automated polyglot persistence on a declarative basis using schema-

annotations. By defining utility functions or performance indices for non-functional requirements like availability and latency as well as demanded binary properties like object-level atomicity, tenants specify their requirements. The mediator scores available backend databases and selects the optimal database for each part of the tenant's schema and automatically routes data and operations. Exploiting the proposed annotations, providers are free to define which requirements they provide SLAs on and users can employ annotations in an opt-in fashion, maximizing flexibility on both ends. We provided evidence that tremendous performance improvements can be expected and outlined the future challenges for providing a general-purpose Polyglot Persistence Mediator.

References

[Agr09] Agrawal, Divyakant et al. Database management as a service: Challenges and opportunities. In *ICDE*, page 1709–1716. IEEE, 2009.

[Bai12] Bailis, Peter et al. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[Bak11] Baker, J. et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, page 223–234, 2011.

[Bas12] Salman A. Baset. Cloud SLAs: Present and Future. *SIGOPS Oper. Syst. Rev.*, 46(2):57–66, July 2012.

[Coo10] Cooper, Brian F et al. Benchmarking cloud serving systems with YCSB. In *SOCC*, page 143–154, 2010.

[Cur11] Curino, Carlo et al. Relational cloud: A database-as-a-service for the cloud. In *Proc. of CIDR*, 2011.

[Dut11] Dutreilh, Xavier et al. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS*, page 67–74, 2011.

[Ges14] Gessert, Felix et al. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB 2014*, 2014.

[Hac02] Hacigumus, H. et al. Providing database as a service. In *ICDE*, page 29–38, 2002.

[Hun10] Hunt, Patrick et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[LS13] Wolfgang Lehner and Kai-Uwe Sattler. *Web-Scale Data Management for the Cloud*. Springer, New York, auflage: 2013 edition, April 2013.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC 14*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[Sob08] Sobel, Will et al. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

Rights and Privacy

Privacy Preserving Record Linkage with PPJoin

Ziad Sehili[1], Lars Kolb[1], Christian Borgs[2], Rainer Schnell[2], Erhard Rahm[1]

[1]University of Leipzig
{sehili,kolb,rahm}@informatik.uni-leipzig.de

[2]University of Duisburg-Essen
{rainer.schnell,christian.borgs}@uni-due.de

Abstract:
Privacy-preserving record linkage (PPRL) becomes increasingly important to match and integrate records with sensitive data. PPRL not only has to preserve the anonymity of the persons or entities involved but should also be highly efficient and scalable to large datasets. We therefore investigate how to adapt PPJoin, one of the fastest approaches for regular record linkage, to PPRL resulting in a new approach called P4Join. The use of bit vectors for PPRL also allows us to devise a parallel execution of P4Join on GPUs. We evaluate the new approaches and compare their efficiency with a PPRL approach based on multibit trees.

1 Introduction

The record linkage process tries to find pairs of entities across different databases that refer to the same real-wold object. Beside the field of data integration, it is increasingly used in research applications, for example in medicine, the social sciences and official statistics. In these fields, protecting the identifiers of the entities is usually required by law. Therefore, if such linkages are permitted, special techniques protecting the identifiers have to be used. The set of techniques for record linkage without revealing identifiers is called *Privacy Preserving Record Linkage* or *PPRL*. Due to the many applications, PPRL is an active field of research in Computer Science, Statistics and some application fields as Epidemiology, Health Service Research and Survey Methodology [Sch15].

Like traditional approaches for record linkage, PPRL has an inherent scalability problem if each (encrypted) record needs to be compared with each other record resulting in a quadratic complexity. The usual means to improve efficiency and thus scalability to larger datasets is to reduce the search space, e.g. by appropriate filter and blocking techniques, or/and to perform record linkage in parallel on many processors [Chr12b, KTR12]. PPJoin (Position Prefix Join) [XWLY08] is an efficient approach for regular record linkage exploiting several filters to reduce the search space; its efficiency has been confirmed in independent evaluation studies for diverse datasets [KTR10, JLFL14]. We will therefore investigate how to adapt this scheme to PPRL where records are encrypted by bit vectors. We will also propose and evaluate the parallel execution of the adapted PPJoin scheme on

graphical processing units (GPUs).

Our contributions are thus as follows

- We present how PPJoin can be adapted to evaluate the similarity of bit vectors for PPRL (Section 4).
- We show how the adapted PPJoin approach, which we call P4Join (Privacy-Preserving PPJoin), can be executed in parallel on GPUs (Section 5).
- We evaluate the efficiency of the new PPJoin approaches (Section 6). The evaluation also includes a comparison with a previously proposed approach based on multibit trees [BRS13].

Before outlining these contributions we begin with preliminaries on the assumed approaches for record linkage and PPRL (Section 2) and a discussion of related work including the multibit tree approach (Section 3). At the end, we summarize and close with an outlook.

2 Preliminaries

For record linkage we apply so-called similarity joins that determine all pairs of records with a similarity above a minimal threshold. We first introduce the notion of such similarity joins together with a simple length filter that can be utilized to reduce the number of necessary comparisons. Furthermore, we introduce the assumed model for encrypting records by bit arrays to support a privacy-preserving record linkage.

2.1 Set similarity joins and length filter

To detect duplicate records between two heterogeneous data sources R and S, similarity joins identify all pairs $(r, s) \in R \times S$ with a similarity above a given threshold t, i.e. $Sim(x, y) \geq t$ for comparable attribute values x and y. The efficient processing of such similarity joins for different kinds of similarity measures has been the focus of much research in the past, e.g., [AGK06, HCKS08, MF12, RLW+13, SHC14, WLF10, XWL08, XWLY08]. In this work we focus on the popular *Jaccard* similarity because it is well suited for both string-tokenized records and bit arrays. Given two records with attribute values x and y, represented as (multi-) sets of tokens (e.g. n-grams), the Jaccard similarity is defined as:

$$Sim_{Jaccard}(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{|x \cap y|}{|x| + |y| - |x \cap y|}. \quad (1)$$

Example: Consider two records $x = tomas$ and $y = tommas$ tokenized to bigrams as shown in Fig. 1. The resulting Jaccard similarity is $6/7$, since all (6) bigrams of x overlap with the 7 bigrams of y.

The similarity function $Sim_{Jaccard}$ allows the application of a simple *length filter* to avoid the evaluation of the Cartesian product to identify all pairs of similar records. This is because the minimal similarity (token overlap) can only be achieved if the lengths of the input records do not deviate too much. Formally, for two records x and y with $|x| \leq |y|$, it holds that

$$Sim_{Jaccard}(x, y) \geq t \Rightarrow |x| \geq \lceil t \cdot |y| \rceil \tag{2}$$

For example, two records cannot satisfy a similarity threshold $t = 0.8$ if their lengths differ by more than 20%. If the first record has, say, ten tokens a comparison is no longer needed for all records with less than eight or more than 12 tokens. The length filter can thus achieve a substantial reduction of the search space and is used in many implementations including PPJoin and Multibit Trees.

2.2 Privacy Preserving Record Linkage

For privacy preservation, we consider the detection of duplicate records in fully encrypted datasets. We therefore encrypt the records' attribute values individually *before* passing them to a semi-trusted third party that run the record linkage process.

To encrypt a single record, we map all n-grams of the (relevant) attribute values to a bit vector (array) of fixed size as proposed in [SBR11]. Specifically, each n-gram (of each n-gram set) is hashed to *multiple* bits by applying k independent hash functions, each defining an index of a bit which is set to one. This can be achieved by a double hashing scheme combining two independent base hash functions f and g to determine the k hash values $h_1(x), \ldots, h_k(x)$ for each n-gram x [KM06]:

$$h_i(x) = (f(x) + i \cdot g(x)) \mod l.$$

As base hash functions, [SBR09] proposed the usage of two keyed hash message authentication codes (HMACs), namely, HMAC-SHA1 and HMAC-MD5 for f and g, respectively. The mapping of records and their n-grams to bit arrays is illustrated in Figure 1 for the two names *tomas* and *tommas*.

The similarity between two records given by the bit arrays (or fingerprints) x and y can now be determined analogously to determining the Jaccard similarity by measuring the degree of overlap between the set bit positions. The resulting similarity is also known as the *Tanimoto* similarity and can be expressed as:

$$Sim_{Tanimoto}(x, y) = \frac{|x \wedge y|}{|x \vee y|} = \frac{|x \wedge y|}{|x| + |y| - |x \wedge y|} \tag{3}$$

with $|x|$ denoting the number of set bits (or cardinality) in the bit array x.

For the example in Figure 1 we have 11 set bits in the intersection of the two bit arrays and 12 in the union resulting in a similarity of $11/12$.

It was shown in [SBR09] that the described encryption scheme is similarity-preserving. It also allows applying the length filter introduced before, i.e. two records can only meet a minimal similarity if their number of set bits does not differ too much.

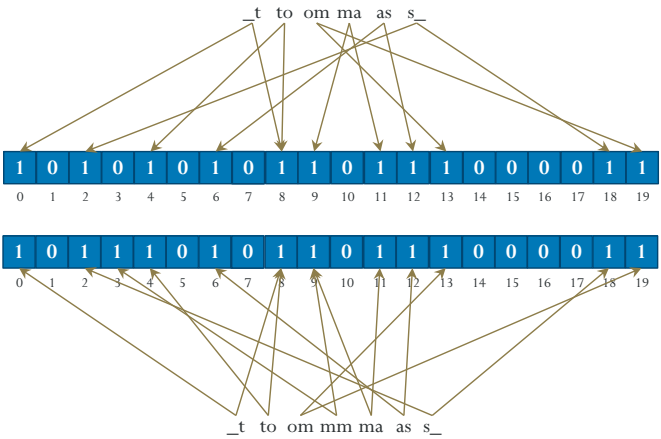


Figure 1: Example of the encryption of two names *tomas* and *tommas*, each tokenized to bigrams, using $k = 2$ hash functions and bit arrays of size 20 bits.

3 Related Work

Record Linkage Record linkage, entity resolution or data matching is the problem of finding similar records referring to the same real-world entity. It has been addressed by numerous research studies and approaches as summarized in several surveys [EIV07, Chr12b, KR10]. Key challenges include achieving complete and accurate results at a good efficiency and scalability. Many approaches aim at improving efficiency by reducing the search space, e.g. by so-called indexing and blocking techniques [DN11, Chr12a]. Furthermore, parallel record linkage approaches have beend devised, e.g. for MapReduce/Hadoop platforms [VCL10, KTR12] or by using graphic processors (GPUs) [FPS+13, NKH+13]. The use of similarity joins [Coh00] for finding all pairs of records with a certain minimal similarity allows several performance improvements by exploiting charactistics of the considered similarity measure and the prespecified similarity threshold. This holds especially for the broad class of so-called *signature-based similarity joins* [AGK06] where the comparison of records is based on their signatures, such as the set of tokens (e.g., n-grams) for selected attributes. In this case, one cannot only utilize the introduced length filter but also exclude all pairs of records that do not share at least one token in their signature. Further proposed optimizations for such similarity joins include the use of a so-called prefix filter and dynamically created inverted indexes [BMS07]. The PPJoin approach [XWLY08] includes these and further optimizations for improved efficiency. Since this algorithm is the basic of our work, it will be presented in detail in Section 4.

Privacy Preserving Record Linkage In [VCV13], a comprehensive taxonomy and survey of privacy-preserving record linkage techniques is provided covering different encryption schemes, linkage and scalability approaches. The approach of [ALM05] uses hashing to map TF/IDF tokens of records to bit vectors for privacy-preserving similarity joins. It

applies a token-based blocking to find candidate pairs that are then compared using the Jaccard similarity function but uses no further filtering to eliminate dissimilar pairs of records.

Similarity Filtering with Multibit Trees The use of multibit trees promises a better efficiency for PPRL and will be considered in our evaluation for comparison purposes. The approach was originally suggested to rapidly query large databases of chemical *fingerprints* (bit arrays) [KNP10]. A query bit array A is being searched for in a database B, with the aim to retrieve all elements in B whose similarity with A is above the threshold t. Multibit Trees are based on the Tanimoto coefficient presented in Eq. 3 as a similarity measure.

As suggested in [BRS13], Multibit trees can easily be used for PPRL with the encryption scheme described in Section 2. If we have two files with records to compare with each other, we can use the larger file to build up the multibit trees and use each record (fingerprint) of the second file for searching similar records. The time taken to build up the index structure is therefore much less important than the query time, which increases linearly with the size of the query file.

The original paper [KNP10] describes several variations and we focus here on the multibit tree scheme that has shown the best evaluation results in [BRS13] and [Sch14] and also outperformed blocking approaches such as Sorted Neighborhood and Canopy Clustering. In this scheme, we partition the fingerprints into buckets according to their lengths such that all fingerprints with the same number of set bits belong to the same partition (or bucket). To apply the length filter, we can then restrict the search for similar fingerprints to the partitions meeting the length criterion of Eq. 2.

Query efficiency is further improved by organizing all fingerprints of a partition within a multibit tree. A multibit tree is a binary tree to iteratively assign fingerprints to its nodes based on so-called match bits. A match bit refers to a specific position of the bit array and can be 1 or 0: it indicates that all fingerprints in the associated subtree share the specified match bit. When building up the multibit tree, one match bit or multiple such bits are selected in each step so that the number of unassigned fingerprints can be roughly split by half ("half clustering" strategy). The split is continued as long as the number of fingerprints per node does not fall under a limit ([KNP10] recommends a limit of 6). The match bits can then be used for a query fingerprint to determine the maximal possible similarity for subtrees when traversing the tree and can thereby eliminate many fingerprints to evaluate.

4 Adaptation of PPjoin for Encrypted Data

PPJoin (Position Prefix Join) [XWLY08] is a signature-based similarity join algorithm that applies several optimizations for improved efficiency, in particular the length filter, a prefix filter and a position filter. We first describe the prefix filter already proposed in [CGK06]. We then outline PPJoin and its adaptation for PPRL.

4.1 Prefix Filter

The prefix filter [CGK06] for signature- or token-based similarity joins exploits that similar records must share at least one token. Depending on the similarity threshold and length of the records, the overlap must even occur for a subset of the tokens. For example, two records with ten tokens each can only meet a minimal similarity of 0.8 if they overlap in at least nine tokens. Hence it is sufficient to check whether subsets of two tokens each overlap; without such an overlap in the subsets the record pair cannot match and safely be eliminated from further consideration. To maximize this filter idea one builds the subsets with the least frequent tokens; i.e. one takes the prefix of a record's tokens ordered by their overall frequency.

Given two records x and y represented as sorted token sets (using the same ordering), the overlap between their tokens is defined as:

$$Overlap(x, y) = |x \cap y| \tag{4}$$

From the similarity join definition using the Jaccard similarity (Eq. 1) we can derive the required minimal overlap in order to meet the similarity threshold t:

$$Sim_{jaccard}(x, y) \geq t \Leftrightarrow Overlap(x, y) \geq \lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil = \alpha \tag{5}$$

Pairs of records with an overlap less than the *minimal overlap* α cannot meet the similarity join condition so that the relatively expensive similarity computation can be saved for them.

The prefix filter uses this observation to determine prefixes of sufficient size so that they have to overlap to meet the minimal similarity. This is the case for prefix lengths $|x| - \alpha + 1$ and $|y| - \alpha + 1$ for x and y, respectively. For the mentioned example above $\alpha = 9$ and the prefixes have to be of length 2.

4.2 Position Prefix Join (PPJoin)

As mentioned before, PPJoin is a set similarity join algorithm using the Jaccard similarity to find *all* similar pairs (x, y), if $Sim_{Jaccard}(x, y) \geq t$. To do this in an efficient way, PPJoin applies length filtering, prefix filtering and position filtering [1]. As in [XWLY08], we describe PPJoin for finding similar records in a single input file R. The more general case with two (duplicate-free) input files can be mapped to this case by merging the two files into one and observing that only records from different input files are compared with each other [VCL10].

In order to apply the length and prefix filters, a preprocessing step is executed first followed by the actual filtering and record comparisons.

[1] [XWLY08] also proposed the use of a suffix filter but this filter is not considered here due to its likely high overhead for encrypted data.

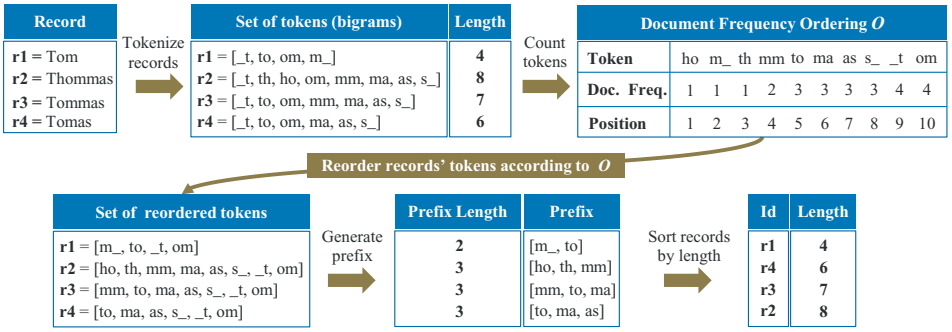


Figure 2: Preprocessing step: The records are first tokenized (to bigrams) and the tokens' frequencies are determined. The resulting document frequency ordering O is then used to order the tokens of the records according to their frequency. After the computation of their prefixes for a threshold $t = 0.8$, the records are sorted by ascending length.

(a) Preprocessing: The main goals of preprocessing are to determine the lengths and prefixes of all records and to sort the records according to their lengths. Determining the prefixes requires to determine the tokens' frequencies and reorder the tokens per record. These tasks are achieved in two passes. First PPJoin reads the records sequentially, determines their lengths and counts the occurrence of each token in *all* the records. The tokens are then sorted by ascending frequency which corresponds to the *document frequency ordering* in IR. In the second pass, the tokens of each record are sorted according to the overall frequency, i.e. from the least frequent to the most frequent token. Furthermore for each record x, the prefix $pref(x)$ is computed. According to Eq. 5, the minimal overlap and thus the prefix length also depends on the length of the record y to which x is to be compared. Since these other records are yet unknown and of varying length, PPJoin applies the following safe prefix length retaining all true matches:

$$|pref(x)| = \lceil (1-t) \cdot |x| \rceil + 1 \tag{6}$$

At the end of preprocessing the records are sorted by length to support the use of the length filter in the next step.

Figure (2) illustrates the preprocessing for an input of four records $r_1 = Tom$, $r_2 = Thommas$, $r_3 = Tommas$ and $r_4 = Tomas$ and a Jaccard similarity threshold $t = 0.8$.

(b) Comparison of records: In the second step, PPJoin jointly applies the length and prefix filters while processing the records in ascending order of their lengths. To quickly determine the records with overlapping prefixes, PPJoin builds up an inverted index I on-th-fly recording per token all (relevant) records having this token in their prefix.

We explain this process by using the four records returned by the preprocessing shown in Figure 2. The first (shortest) record r_1 has length 4 and prefix $pref(r_1) = [m\_, to]$. The index I is initialized with the two tokens pointing to record $r1$ (Figure 3a). The next record r_4 has length 6 and prefix $pref(r_4) = [to, ma, as]$. The length filter check reveals that $|r_1| < \lceil t \cdot |r_4| \rceil$ $(4 < \lceil 0.8 \cdot 6 \rceil)$ so that $r1$ does not have to be compared against $r4$ and any further records which have at least length 6. Hence, when adding the new

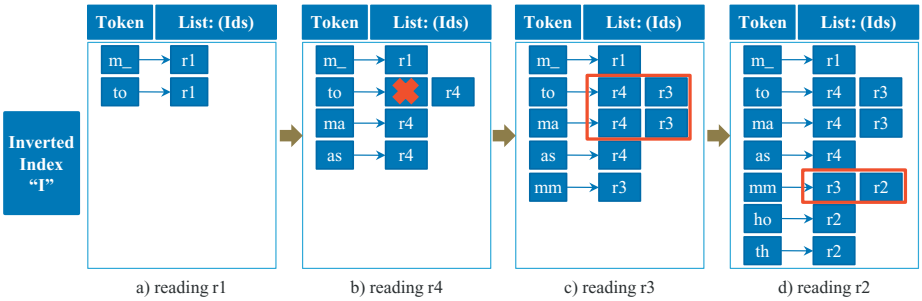


Figure 3: Index usage for PPJoin

prefix tokens to the index, all index entries found for $r1$ can be safely deleted as shown in Figure 3b. When processing the further records r_3 and r_2 the inverted index reveals the prefix overlaps between r_4 and r_3 and between r_3 and r_2 (shown in red squares in Figure 3c and 3d respectively) so that the respective records need to be compared with each other.

PPJoin makes additional use of the prefix tokens by a so-called *position filter*. To approximate the maximal possible overlap between two records, it considers the number of common prefix tokens as well as the position of the last common prefix token. With this information one can derive the maximal possible overlap by assuming that all non-prefix tokens overlap to the maximal extent possible. For example, records r_3 and r_2 in Figure 3d, share only one prefix token, mm, at positions $pos(mm)_{r3} = 1$ and $pos(mm)_{r2} = 3$. This last common token separates each record into a left part lp representing the tokens already seen and a right part rp of unseen tokens. Furthermore the length of rp equals the length of the record minus the position of the last common token, e.g. $rp(r_3) = |r_3| - pos(mm) = 7 - 1 = 6$ and $rp(r_2) = 8 - 3 = 5$. The right parts can thus overlap in at most 5 tokens; together with the prefix overlap in the left parts ($= 1$) the total overlap between r_2 and r_3 is at most $5 + 1 = 6$.

In general, the maximal overlap between two records x and y can be determined by

$$MaxOverlap(x, y) = |lp(x) \cap lp(y)| + min(|rp(x)|, |rp(y)|)$$

The record pair can be filtered out from the similarity comparison if this maximal overlap is smaller than the minimally needed overlap α given in Eq. 5. This is the case for our example because $MaxOverlap(r_3, r_2) = 6 < \alpha = 7$.

4.3 PPJoin for Encrypted Data (P4Join)

For the PPRL version of PPJoin, called P4Join (Privacy-Preserving Prefix Position Join), all input records are encrypted as same-sized bit arrays as explained in Section 2.2. Many parts of PPJoin can then be adapted rather easily by considering the set bit positions (indexes) as the new "tokens". The length of a record thus corresponds to the number of set

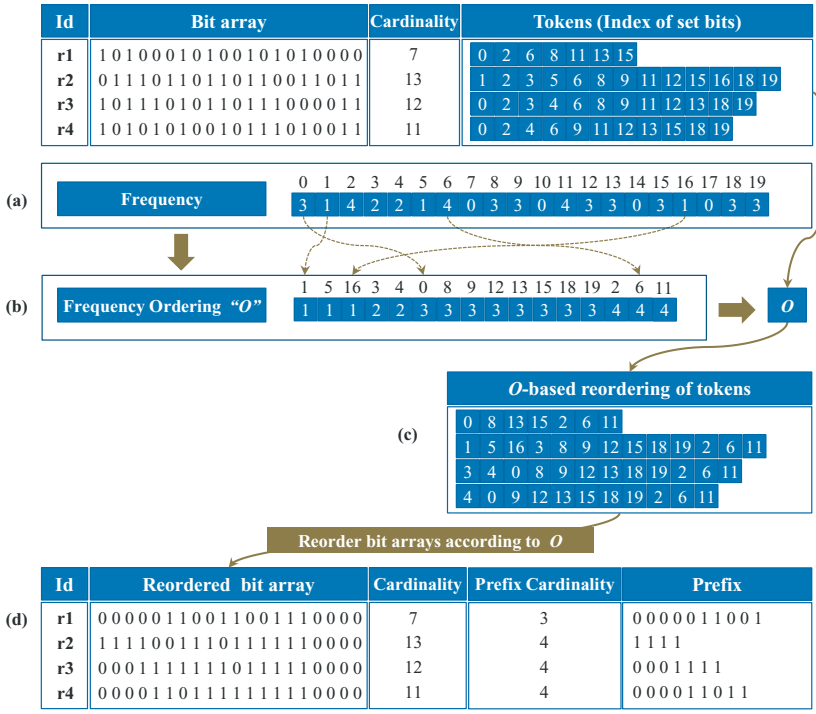


Figure 4: P4Join preprocessing. The top part shows the computation of the frequency ordering O and the bottom part the reordering of the bit arrays and the computation of prefixes for $t = 0.8$.

bits in its bit array which we call its *cardinality* to avoid confusion. We first explain the adapted preprocessing phase and then the use of the filters in the main processing phase. As we will see, P4Join can be realized without building an inverted index. We will also devise an improved position filter.

(a) Preprocessing: As before, preprocessing works in two passes. In the first pass we determine for each index position its frequency, i.e. we count for how many records it is set. The index positions are then ordered in ascending frequency resulting in an ordering O. Figure 4a and 4b show an example of the position frequencies and the generation of O for four records that are encrypted by bit arrays of size 20. Index positions with frequency 0 (positions 7, 10, etc.) are unset in all bit arrays and can simply be ignored in the further processing. In the second pass, the bits per bit array are reordered according to O as shown in Figure 4c. The prefix length is again calculated according to Eq. 6. Again, the length refers to the number of set positions (cardinality) and not to the size of the prefix. For example in Figure 4d the prefix of record r_1 contains *all* starting bits of the reordered r_1 bit array until we reach the prefix's cardinality 3. Hence, prefixes of the same length may have different sizes.

(b) Comparison of bit arrays: In this second step we process records differently than for PPJoin. This is because we observed in preliminary experiments that the overhead to

Algorithm 1: P4Join (without preprocessing)

Input : List of records (bit array) R sorted by ascending cardinality;
Similarity threshold t;
Output: $\{(x, y) \in R \times R \mid Sim(x, y) \geq t\}$

```
1  result ← [];
2  lmap ← new Map(length: List<record>);
3  foreach x ∈ R do
4      foreach e ∈ lmap do
5          if e.length < |x| · t then                                  // Length filter
6              lmap.remove(e);
7          else
8              foreach record y ∈ e.List do
9                  if pref(x) ∧ pref(y) ≠ 0 then                       // Prefix filter
10                     if not Positional_Filter(x, y) then
11                         if (Sim_Tanimoto(x, y) ≥ t) then
12                             result.add(x, y);
13     lmap.add(x);
14 return result;

15 Boolean Positional_Filter(x, y)
16     prefOverlap ← |pref(x) ∧ pref(y)|;
17     p1 ← position of the last set bit in pref(x);
18     p2 ← position of the last set bit in pref(y);
19     diff1 ← 0;
20     diff2 ← 0;
21     if p1 > p2 then
22         diff1 ← number of set bits in pref(x) having position > p2;
23     else
24         diff2 ← number of set bits in pref(y) having position > p1;
25     restOverlap ← min[(|x| − |pref(x)| + diff1), (|y| − pref(y) + diff2))];
26     MaxOverlap ← prefOverlap + restOvelap;
27     return MaxOverlap < ⌈(|x| + |y|) · t/(1+t)⌉;          // MaxOverlap < MinOverlap?
```

maintain and use an inverted index is too high compared to the achievable savings (which are lowered by the fact that the actual similarity computation is much faster for bit arrays than for general sets of tokens). We similarly observed that using the original position filter of PPJoin results in a performance degradation so that we devise a new position filter.

Algorithm 1 shows the pseudo-code of the main phase of P4Join using as input the output of the preprocessing step as well as the similarity threshold t. It uses a data structure $lmap$ (Line 2) that lists for each record length (cardinality) the records that are still relevant for comparisons based on the length filter. While the shown algorithm uses all three supported filters (length, prefix, position) we can easily deselect some of these filters to evaluate their relative performance impact (this will be used in the evaluation). We now discuss the use and implementation for each of the filters.

Length Filter: According to the shown algorithm, all records are read sequentially in the given order with ascending cardinality (see top of Figure 5). When a record x with cardinality c is read, it is added to $lmap$ either together with a new entry for cardinality c or by appending it to an existing record list for c (Line 13). The first record is immedi-

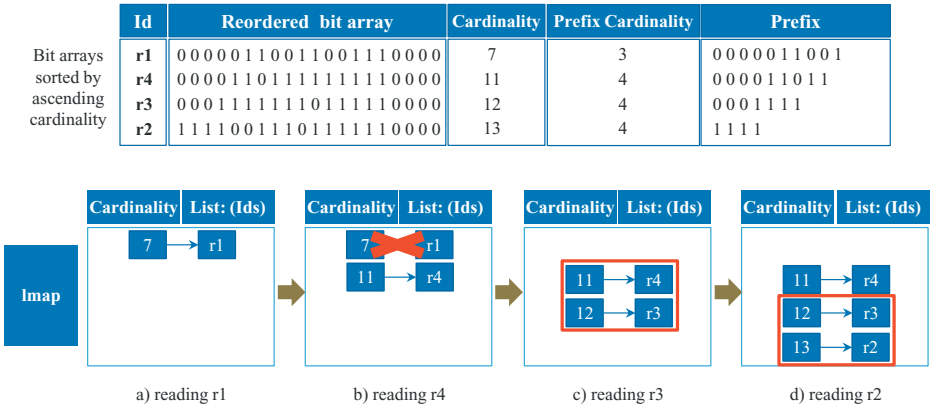


Figure 5: P4Join processing for four sample records

ately added while for a non-empty $lmap$ the addition follows after applying the length filter (and possibly further filters) and performing the necessary similarity checks against already read records. For example, reading the first record r_1 results in the state of $lmap$ shown in (Figure 5a) with r_1 listed for cardinality 7.

The $lmap$ data structure makes it easy and efficient to identify candidates that no longer need to be considered according to the length filter. This is achieved by the second $foreach$ loop where we check for each length in $lmap$ whether the length filter applies in comparison with the current record x. Each record with such a length can safely be excluded from all further comparisons since all further records have the same or larger cardinality than x (Lines 4 to 6). When processing the second record r_4 we therefore determine that cardinality 7 does not satisfy the length filter for threshold $t = 0.8$ ($7 < \lceil 0.8 \cdot 11 \rceil$). Hence this entry together with its record $r1$ can be safely deleted from $lmap$ (Figure 5b). After eliminating the entry, record r_4 is added to a new $lmap$ element for cardinality 11. The further records, r_3 and r_2, do not allow a similar reduction so that all records need to be compared with each other if we would only apply the length filter.

Prefix Filter: The prefix filter avoids the comparison for two records if their prefixes do not share at least one set bit position. In PPJoin the prefix overlap is checked with an inverted index which is relatively expensive to maintain and use for bit arrays. This is because the bit arrays make it very cheap to directly determine the overlap by simply computing the `AND` operation between the two prefixes. Only if the resulting bit array has at least a single bit set we need to further consider the pair of records (Line 9); otherwise we can avoid the similarity computation. Figures 5c and 5d show in red squares the two pairs that pass the prefix filter. For example the records r_4 and r_3 pass the length filter ($11 > \lceil 0.8 \cdot 12 \rceil$) and the prefix filter (`000011011 AND 0001111 = 000011000`), so that they must be compared with each other (unless the position filter avoids this which will be the case, see below). By contrast, records r_4 and r_2 have no common bit position in their prefixes so that their similarity comparison is saved.

Position Filter: Applying the PPJoin position filter to bit arrays turned out to be rela-

tively expensive (partly due to the need to determine and use the positions of last common prefix tokens) but was also limited by an overly imprecise estimate for the maximal overlap. We can determine a more accurate maximum overlap between two records by considering the *last prefix tokens* instead of the *last common prefix token*. We explain the idea for records r_4 and r_3 having cardinalities 11 and 12 and the prefixes of set positions $pref(r_4) = [4, 5, 7, 8]$ and $pref(r_3) = [3, 4, 5, 6]$ as shown in the top of Figure 5. The overlap between the two prefixes equals 2 and the last common prefix token, 5, has the positions $pos(5)r_4 = 2$ and $pos(5)r_3 = 3$. Using the PPJoin position filter, we thus obtain $MaxOverlap(r_4, r_3) = 2 + min(11 - 2, 12 - 3) = 11$. This overlap is not smaller than the minimum required overlap $\alpha = \lceil (11 + 12) \cdot (0.8/1.8) \rceil = 11$ (Eq. 5) so that the similarity computation between r_4 and r_3 cannot be saved.

However, we observe that token 6 in $pref(r_3)$ cannot contribute to $MaxOverlap$ because it refers to a bit position smaller than some tokens in $pref(r_4)$, namely 7 and 8. In general, given two prefixes $pref(x)$ and $pref(y)$ having as last tokens a and b respectively, if $a < b$ then the only prefix tokens that can contribute to $MaxOverlap$ (beside the common tokens) are those tokens from $pref(y)$ greater than a. For r_3, the maximal number of possibly overlapping tokens is thus the number of common prefix tokens (2) plus the number of non-prefix tokens (8), i.e. 10 in total. For r_4, the maximal possible overlap is the number of common prefix tokens (2), plus the number of non-prefix tokens (7) plus the number of tokens in $pref(r_4)$ greater than the last token 6 in $pref(r_3)$ (= 2), i.e. 11 in total. The maximal overlap is the minimum of 10 and 11 which is 10 and below the minimal overlap $\alpha = 11$. Hence, we can filter out r_4 and r_3 from the similarity comparison.

This new position filtering can thus prune more dissimilar pairs of records compared to the original PPJoin position filter. Its implementation as a function is shown in the bottom part of Algorithm 1. It determines the last tokens in the two prefixes. The smaller of the two determines how many tokens of the other prefix may contribute to $MaxOverlap$. The function returns the boolean value true if the maximal overlap is smaller than the minimally needed overlap α. Hence in the main algorithm the similarity comparison is only performed if the position filter does not apply (Line 10).

5 Matching Encrypted Data with GPUs

The utilization of Graphical Processing Units (GPUs) to speed-up similarity computations is a comparatively new approach [FPS+13]. Modern GPUs provide thousands of cores that allow for a massively-parallel application of the same instruction set to disjoint data partitions. The availability of frameworks like CUDA[2] and OpenCL[3] simplify the utilization of GPUs to parallelize general purpose algorithms. In this work, we rely on OpenCL which, in contrast to CUDA, is supported by different hardware vendors. An OpenCL-Program, which is also referred to as a *Kernel*, is written in a dialect of the C programming language. The number of kernels that can be applied to disjoint data partitions simultaneously de-

[2] `https://developer.nvidia.com/category/zone/cuda-zone`
[3] `https://www.khronos.org/opencl/`

pends on the number of cores provided by the GPU and the memory requirements of the Kernel instances.

Compared to developing "CPU programs", GPUs and their programming languages like OpenCL have several limitations. Before and after the computation, input and output data must be transferred between the main memory of the host system and the memory of the GPU. Furthermore, on GPUs dynamic memory allocation is not possible at runtime, i.e., the resources required by an algorithm must be known and allocated a priori. Usually the memory capacity of GPUs is smaller than the available main memory. This requires to divide the overall workload into smaller tasks which are executed by the GPU in multiple rounds. Thereby, the scheduling of the individual tasks should minimize the data volume to be transferred between host system and GPU. Furthermore only basic data types (e.g., int, long, float) and fixed-length data structures (e.g., arrays) can be used. Despite such limitations, the utilization of GPUs is a promising approach to speed up record linkage on encrypted datasets. All records are represented as bit arrays of equal length, which can be expressed by arrays of type long (64 bits). Furthermore, the similarity computation can be broken down into simple bit operations which can be easily processed by GPUs.

We describe the GPU version of P4Join for the general case of two input datasets R and S; the special case with only one dataset R is easily supported by comparing R with itself. The preprocessing is performed on the host system while the filtering and similarity checks are performed on the GPUs. Only the length and prefix filters are applied as a GPU implementation of the position filter becomes already too complex compared to the similarity computation itself. We also support a hybrid scheme utilizing both CPUs and GPUs for similarity computation.

5.1 Preprocessing

In a preprocessing step, the records of the input datasets R and S are sorted by their cardinality, i.e., the number of bits set to one. Additionally, the bits of all records are ordered by the document frequency in ascending order. Furthermore, each record is annotated with its cardinality and its prefix fingerprint (see Section 4.3). In general, the input datasets as well as the resulting correspondences exceed the available memory of the GPU. Thus, R and S are range-partitioned into fixed-sized partitions. Similar to [HKGR13], pairs (R_i, S_j) are then iteratively shipped to the GPU for similarity computation. Before a partition pair is shipped to the GPU, it is checked whether the two partitions contain at least one record pair which passes the length filter (see Eq. 2). To this end, the cardinality values of the first and the last record of the two (sorted) partitions are compared. If there is no such record, the pair is skipped. Otherwise it is transferred to the GPU.

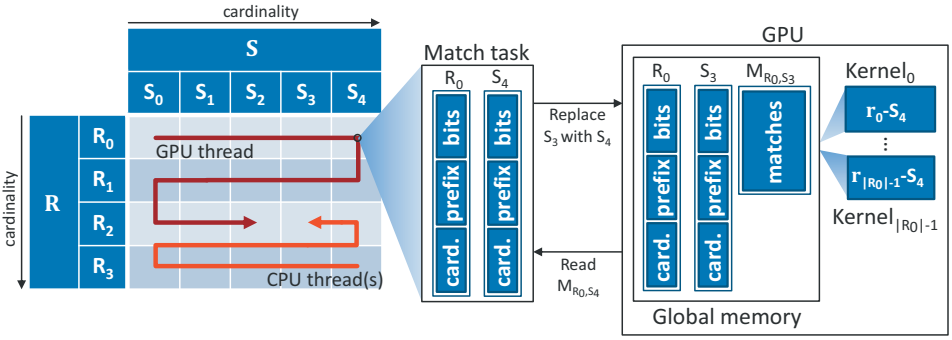


Figure 6: Execution scheme for hybrid GPU/CPU-based record linkage minimizing the data transfer between the host program and the GPU.

5.2 Massively-parallel Record Linkage

The GPU executes a kernel instance for each record $r \in R_i$. A kernel instance is responsible for comparing r with each $s \in S_j$. Thereby, the kernel iterates over all $s \in S_j$ (which are sorted by cardinality) and consecutively checks whether the length filtering constraint is fulfilled and whether the evaluation of subsequent $s' \in S_j$ can be skipped. For each remaining candidate record, it is checked whether the prefix fingerprints do overlap. If this is the case, their similarity is computed according to Eq. 3.

The indexes of all $s \in S_j$ with $Sim_{Tanimoto}(r, s) \geq t$ are written to a kernel instance-specific range of an output array of type int which can be accessed by all kernel instances. The partial results are later unified by the host program.

To minimize the data volume to be transferred between the main memory and the GPU we also support a *hybrid CPU/GPU execution scheme* similar to [HKGR13]. We utilize a task queue that supports the parallel matching of different partition pairs on both the GPU as well as on the CPU. A dedicated thread takes tasks from this queue and submits them to the GPU. In addition to this GPU thread, several CPU threads can access the job queue from the opposite end to independently perform matching on the CPU. We select jobs and ship partitions using the scheme displayed in Figure 6. This scheme ensures that after completion of a GPU job only one new partition need to be transferred to it. The other partition remains in the GPU's memory and is reused for the next job.

6 Evaluation

We evaluate the proposed P4Join approach and its filters for several datasets of different sizes and compare the resulting execution times with the use of the Multibit tree algorithm and a naive nested loop implementation. We also evaluate the GPU implementation of P4Join. Before presenting the evaluation results we introduce the experimental setup

n	100,000	200,000	300,000	400,000	500,000
|R|= n/5	20,000	40,000	60,000	80,000	100,000
|S|= 4 * n/5	80,000	160,000	240,000	320,000	400,000

Figure 7: Size of the datasets (number of records) used in the evaluation

including the used datasets.

Note that for a given dataset all the algorithms produce the same match result, i.e. they find *all* the pairs having a similarity greater or equal to a predefined threshold. Hence, the quality of the match result does not depend on the considered algorithms but on the input datasets and the encryption method. The quality of the used encryption method has already been evaluated in [SBR09, BRS13].

6.1 Experimental setup

In our experimental setup we largely follow the settings used in previous evaluations on privacy-preserving record linkage such as [BRS13]. We utilize the data generator from [Chr05] to generate five differently sized datasets of person records. Each dataset consists of n records that are assigned to two subsets R and S of size $1/5 \cdot n$ and $4/5 \cdot n$ respectively such that R contains duplicate records of records in S (see Figure 7). The records are tokenized into bigrams which are mapped to bit vectors of length $l = 1000$ using $k = 20$ hash functions. For each configuration, we apply a Jaccard similarity threshold of $t = 0.8$. For the Multibit tree experiments, we use a Java implementation of the approach[4] with the clustering strategy *split in half* and the minimal node size of 6 (see Section 3).

All configurations of the first experiment are conducted single-threaded on a desktop machine with a 4-core $2.67GHz$ CPU and 4GB of main memory. For the second experiment, we utilize two low-profile GPUs, namely a Nvidia GeForce GT 610 (48 CUDA cores@810MHz, 1GB memory, 35 €) and a Nvidia GeForce GT 540M (96 CUDA cores @672MHz, 1GB memory). For the GPU-based comparisons, the datasets R and S are split into partitions with a maximum size of $2,000$ records.

6.2 Comparing P4Join with Multibit Trees and NestedLoop

Figure 8 shows the achieved execution times for record linkage on the five datasets for a naive nested loop implementation (evaluating the Cartesian product), the previously proposed Multibit tree as well as the P4Join and its filters. As expected the execution times increase substantially (almost quadratically) with the dataset size. The Multibit Tree approach consistently outperforms NestedLoop albeit only to a relatively small degree of less

[4] http://www.birc.au.dk/~tgk/TanimotoQuery/

Algorithm	Dataset size				
	100,000	200,000	300,000	400,000	500,000
Nested loop	6.10	27.68	66.07	122.02	194.77
Multibit Tree	4.68	18.95	40.63	78.23	119.73
P4Join, length filter only	3.38	20.53	46.48	88.33	140.73
P4Join, length + prefix filter	3.77	22.98	52.95	99.72	159.22
P4Join, all filters	2.25	15.50	40.05	77.80	125.52

Figure 8: Runtime in minutes for P4Join with different filters compared with the nested loop and multibit tree approaches

than a factor of 2. This shows already that the applied filtering and reduction of the search space are not so significant if the similarity computations are cheap which is the case for computing the Taminoto similarity on bit arrays. This also limits the effectiveness of the P4Join implementation for which we separate the results based on the use of the different filters.

The best results are achieved by applying all filters including the new position filter. In this case we achieve similarly good results than for the Multibit Tree. Most of the savings in execution time compared to NestedLoop are already achieved by the length filter which can exclude many comparisons with a simple length check. By contrast the prefix and position filters incur a check per record pair which may be unsuccessful and at best saves a single comparison which is not much more expensive for bit arrays than the filter check. This was especially a limitation for the prefix filter that did not pay off in combination with the length filter. To explain this somewhat surprising result we checked more closely the record and prefix characteristics. We observed that for our settings the generated bit arrays have an average cardinality of 300 and an average prefix length of 60. Such large prefixes lead to a high probability of non-empty intersections so that relatively few comparisons could be saved while the overhead of the prefix check occurs for every record pair passing the length filter.

P4Join preprocessing was generally very fast compared the whole execution time. It varied from $6s$ for the smallest dataset ($100,000$ records) to $24s$ for the largest ($500,000$ records),i.e. less than 0.3% of complete execution time.

6.3 GPU-based results

The relatively slow improvements in execution time show the need for more optimizations such as the use of parallel processing. As outlined in Section 5, the simplicity of P4Join made it possible to develop a parallel implementation for GPUs that we evaluate now.

Figure 9 shows the achieved execution times for the five datasets on the two considered graphic cards. We also present results for the hybrid case when we use three CPU threads

Graphic Card	Dataset size				
	100,000	200,000	300,000	400,000	500,000
GeForce GT 610	0.33	1.32	2.95	5.23	8.15
GeForce GT 610 + 3CPUs	0.30	1.15	2.57	4.50	7.03
GeForce GT 540M	0.28	1.08	2.41	4.28	6.67
GeForce GT 540M + 3CPUs	0.22	0.87	1.95	3.45	5.43

Figure 9: Runtime in minutes of two different graphic cards, also in hybrid mode with 3 CPU threads.

for similarity computation in addition to the GPUs. We observe that both GPUs allow huge improvements in efficiency by reducing execution times by more than a factor 10 and a factor 15-20 for the largest dataset compared to the sequential CPU execution of P4Join. The hybrid approach allows a further improvement by 10-20% by utilizing the CPU threads for additional parallelism and saved data transfers to the GPU. For the largest dataset we could thus improve the execution time to only about 5 minutes compared to 125 minutes for the sequential P4Join execution and 195 minutes for NestedLoop.

The results show the high potential of executing P4Join in parallel and that both GPU and CPU parallelism can be effectively combined.

7 Conclusions and Future Work

We showed how the PPJoin approach for similarity joins can be adapted to privacy-preserving record linkage where sensitive records are encrypted by bit arrays. The new approach called P4Join supports a length, prefix and an optimized position filter. We also showed how P4Join can be executed in parallel on GPUs. Our evaluation revealed that the efficient similarity computation for bit arrays reduces the optimization potential for filter techniques such as the ones in PPJoin and other similarity join implementations. Still the proposed P4Join approach and especially the length filter and the new position filter proved to be effective in reducing the execution time. The biggest performance gains are achieved by the parallel computation on GPUs with a speedup of up to 20 even for low-profile graphic cards.

We see a strong need for further research on improving the efficiency of PPRL schemes to support their scalability to very large datasets. First, the almost quadratic increase of execution times w.r.t input size needs to be improved, e.g. by the use of tailored blocking mechanisms. Furthermore, parallel processing should be employed more comprehensively, in particular on clusters of processing nodes in addition to the node-specific parallel record linkage using GPUs and several CPU cores.

References

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 918–929, 2006.

[ALM05] Ali Al-Lawati, Dongwon Lee, and Patrick McDaniel. Blocking-Aware Private Record Linkage. In *Proceedings of 2nd International Workshop on Information Quality in Information Systems*, pages 59–68, 2005.

[BMS07] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web*, pages 131–140, 2007.

[BRS13] Tobias Bachteler, Jörg Reiher, and Rainer Schnell. Similarity Filtering with Multi-bit Trees for Record Linkage. Technical Report WP-GRLC-2013-01, German Record Linkage Center, 2013.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[Chr05] Peter Christen. Probabilistic Data Generation for Deduplication and Data Linkage. In *Proceedings of 6th International Conference on Intelligent Data Engineering and Automated Learning*, pages 109–116, 2005.

[Chr12a] Peter Christen. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, 2012.

[Chr12b] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-centric systems and applications. Springer, 2012.

[Coh00] William W. Cohen. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM Transactions on Information Systems*, 18(3):288–321, 2000.

[DN11] Uwe Draisbach and Felix Naumann. A Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In *Proceedings of 5th International Conference on Data and Knowledge Engineering*, pages 18–24, 2011.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[FPS+13] Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. Duplicate Detection on GPUs. In *Proceedings of the 15th Conference on Database Systems for Business, Technology, and Web*, pages 165–184, 2013.

[HCKS08] Marios Hadjieleftheriou, Amit Chandel, Nick Koudas, and Divesh Srivastava. Fast Indexes and Algorithms for Set Similarity Selection Queries. In *Proceedings of the 24th International Conference on Data Engineering*, pages 267–276, 2008.

[HKGR13] Michael Hartung, Lars Kolb, Anika Groß, and Erhard Rahm. Optimizing Similarity Computations for Ontology Matching - Experiences from GOMMA. In *Proceedings of the 9th International Conference on Data Integration in the Life Sciences*, pages 81–89, 2013.

[JLFL14] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String Similarity Joins: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 7(8):625–636, 2014.

[KM06] Adam Kirsch and Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *Algorithms - ESA 2006, 14th Annual European Symposium*, pages 456–467, 2006.

[KNP10] Thomas Greve Kristensen, Jesper Nielsen, and Christian N. S. Pedersen. A tree-based method for the rapid screening of chemical fingerprints. *Algorithms for Molecular Biology*, 5:9, 2010.

[KR10] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[KTR10] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[KTR12] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[MF12] Ahmed Metwally and Christos Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[NKH+13] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *Proceedings of the 10th International Extended Semantic Web Conference*, pages 275–289, 2013.

[RLW+13] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. Efficient and Scalable Processing of String Similarity Join. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[SBR09] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Privacy-preserving record linkage using Bloom filters. *BMC Med. Inf. & Decision Making*, 9:41, 2009.

[SBR11] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. A Novel Error-Tolerant Anonymous Linking Code. Technical Report WP-GRLC-2011-02, German Record Linkage Center, Duisburg, 2011.

[Sch14] Rainer Schnell. An efficient privacy-preserving record linkage technique for administrative data and censuses. *Statistical Journal of the IAOS*, 30(3):263–270, 2014.

[Sch15] Rainer Schnell. Privacy Preserving Record Linkage. In Katie Harron, Harvey Goldstein, and Chris Dibben, editors, *Methodological Developments in Data Linkage*. Wiley, 2015. To appear.

[SHC14] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *Proceedings of the VLDB Endowment*, 7(12):1059–1070, 2014.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the International Conference on Management of Data*, pages 495–506, 2010.

[VCV13] Dinusha Vatsalan, Peter Christen, and Vassilios S. Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.

[WLF10] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. *Proceedings of the VLDB Endowment*, 3(1):1219–1230, 2010.

[XWL08] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944, 2008.

[XWLY08] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient Similarity Joins for Near Duplicate Detection. In *Proceedings of the 17th International Conference on World Wide Web*, pages 131–140, 2008.

D2P*t*: Privacy-Aware Multiparty Data Publication

J. H. Nielsen, D. Janusz, J. Taeschner, J.-C. Freytag

Database and Information Systems Group
Humboldt-Universität zu Berlin
{nielseja,janusz,taeschnj,freytag}@informatik.hu-berlin.de

Abstract: Today, publication of medical data faces high legal barriers. On the one hand, publishing medical data is important for medical research. On the other hand, it is neccessary to protect peoples' privacy by ensuring that the relationship between individuals and their related medical data remains unknown to third parties. Various data anonymization techniques remove as little identifying information as possible to maintain a high data utility while satisfying the strict demands of privacy laws.

Current research in this area proposes a multitude of concepts for data anonymization. The concept of k-anonymity allows data publication by hiding identifying information without losing its semantics. Based on k-anonymity, the concept of t-closeness incorporates semantic relationships between personal data values, therefore increasing the strength of the anonymization. However, these concepts are restricted to a centralized data source.

In this paper, we extend existing data privacy mechanisms to enable joint data publication among multiple participating institutions. In particular, we adapt the concept of t-closeness for distributed data anonymization. We introduce *Distributed two-Party t-closeness (D2Pt)*, a protocol that utilizes cryptographic algorithms to avoid a central component when anonymizing data adhering the t-closeness property. That is, without a trusted third party, we achieve a data privacy based on the notion of t-closeness.

1 Introduction

Over the past years, various incidents of *privacy breaches* have fueled the growing demand for preserving privacy of individuals. The ever increasing digital footprint of individuals makes it easier to gather *sensitive private information*. Abuses in the past show that the collection of data sets imposes a risk for revealing sensitive private information [Swe97]. On the other hand, data collection and exchange is necessary to support the needs of institutions that rely on it, e. g., medical institutions conducting a clinical trial. Therefore, taking privacy protection approaches to an extreme by not releasing data at all is no solution. Only revealing non-sensitive information does not provide a feasible solution as well, since medical institutions rely on information about sensitive data, e. g., diseases. The challenge is to find a balance between the privacy interests of individuals and the interests of organizations and companies to gain access to personal data.

Research in the area of privacy protection has developed the domains of Privacy-Preserving Data Mining (PPDM) and Privacy-Preserving Data Publishing (PPDP) [CT13]. Both areas

aim to allow access to data sets while preserving privacy of individuals. However, both areas differ in how to handle data sets. The focus of PPDM is to answer queries targeting disclosed data sets in a privacy preserving way. The focus of PPDP is to generate and publish a complete data set that preserves the privacy of individuals. This paper will focus on microdata protection techniques of PPDP.

One major field of application for PPDP are clinical trials. Clinical trials rely on the publication of private data. Especially, in the field of medicine data is considered highly sensitive. The implied trade-off becomes hard to solve. First, to maintain the utility of the data. Second, to keep the data private at the same time. Therefore, the essential idea when publishing clinical trial data is to achieve a minimal information loss without violating the privacy of individuals.

The Health Insurance Portability and Accountability Act (HIPAA) [US96] is a legislative effort to protect individuals when releasing information about them. For example, HIPAA specifically addresses the removal of personally identifiable information when publishing medical data of individuals (also known as the HIPAA Privacy Rule and the HIPAA Safe Harbor Provision). The notion of privacy and sensitive information are considered to be either to rigid or to imprecise to protect the privacy of individuals [LeF07].

To overcome the limitations imposed by the law, different concepts have been proposed for PPDP. The concept of k-*anonymity* seeks to protect sensitive private information by altering specific attributes using syntactical rules [Swe02b]. An enhancing notion of privacy is the concept of t-*closeness*, enriching the concept of k-anonymity with the semantic relationship between attribute values [LLV07]. However, both concepts operate on a single data set.

Generally, privacy concepts that consider sensitive attributes (SAs) are limited to a single sensitive attribute [FAN11]. In contrast, multi-center clinical trials carried out by more than one institution rely on the publication of data containing more than one SA. Each institution collects different data about the same patient, thus forming a *vertically partitioned* data collection. Such a scenario creates the *multiple sensitive attributes (MSA)* problem [FAN11], i. e., multiple sensitive attributes collected by multiple institutions. Several approaches recognize the importance of MSA without providing necessary algorithms tailored to the specific needs of a distributed data collection environment [DB12].

In the light of this discussion, the contribution of this paper is threefold: (*1*) We extend the concept of t-closeness to a distributed vertically partitioned environment with multiple sensitive attributes and analyze the demands imposed on the concept of t-closeness, (*2*) we develop a protocol named *D2Pt*, which achieves data anonymization utilizing the concept of t-closeness and (*3*) we present an experimental evaluation of our protocol D2P*t*. Furthermore, we compare D2P*t* to previous state-of-the-art protocols that do no utilize semantic data anonymization.

The remainder of this paper is organized as follows. In section 2, we provide background information on data anonymization techniques following the notion of PPDP. In section 3, we present our extended concept of t-closeness and introduce our protocol D2P*t*. In section 4, we show our evaluation results as well as an analysis of our results. In section 5, we provide an overview on related work. We conclude and outline future work in section 6.

2 Background

Nowadays, data publication focuses on publishing data as *microdata*, typically in tabular form [CDCdVFS07]. Microdata contains unaltered data statistics on individuals [Dal77]. An example of a microdata table can be found in Table 1a.

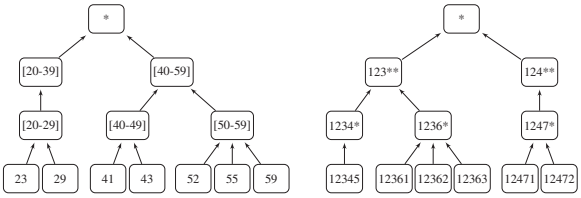
2.1 Centralized data-anonymization

Attributes contained in a microdata-table can be classified into three categories: *identifiers (IDs)*, *sensitive attributes (SAs)* and *quasi-identifiers (QIDs)*. The ID uniquely identifies an individual. An example is the full name of an individual. The SA specifies an attribute an individual does not want to be associated to, e. g., a disease. For that reason the ID is removed from a microdata publication. This process is called *de-identification* [Swe97]. In [Swe97] Sweeney showed that de-identification does not suffice to protect microdata publications. This is due to other publicly available data sets, like voter lists. Voter lists contain the name as well as demographic information like the date of birth, age and gender of an individual. The combination of these attributes values is often unique [Swe97], hence they are termed QIDs. The de-identified microdata table can be *linked* to the publicly available data by joining them on the QID.

k-anonymity To protect against linking, Sweeney introduced the concept of *k-anonymity* in [Swe02b]. Intuitively, k-anonymity alters the values of the QIDs, thus making every individual in that microdata indistinguishable from at least $k-1$ individuals w. r. t. to QID values [Swe02b]. The values are not randomly altered, rather they are being removed or replaced by a "less specific but semantically consistent value" [Swe02a]. This mechanism is called *generalization and suppression* [FED94]. A group of individuals in a microdata table is said to form an *equivalence class (EC)* if they equal in their QID values.

A semantically consistent value is determined by the use of a *value generalization hierarchy (VGH)* [Sam01]. A VGH is a directed, balanced tree describing the proper generalization of attribute values. The specific values are located in the leaves of the tree. the more general values are located in the inner nodes. The root node consists of the most general value or the value '*', which indicates the suppression of a value. A VGH for the attribute AGE can be found in Figure 1a. An example for the generalization of the value 22 of attribute AGE is the age span $[20-29]$.

The Datafly-algorithm Meyerson et al. and Aggarwal et al. have shown that the problem of an optimal anonymization is NP-hard [MW04, AFK$^+$05]. Therefore, approximate solutions to achieve k-anonymity exist. In [Swe97] Sweeney suggested the *Datafly*-algorithm. In its basic version the Datafly-algorithm uses *full-domain generalization* without suppression, i. e., at every state of the algorithm each QID value is at the same level of the VGH. In every iteration of the algorithm, Datafly chooses one out of the QID attributes, specifically the attribute containing the most different values. The algorithm then



(a) VGH for QID AGE (b) VGH for QID ZIP

Figure 1: Value generalization hierarchies

generalizes the values of that attribute, i. e., replacing every occurrence of a value with the value one level up in the VGH. It continues until every individual in the microdata table is indistinguishable from at least $k-1$ other individuals in that table w. r. t. QID values. This algorithm will serve as an example throughout this paper, therefore, we will elaborate more and show its operating principle using an example.

Example 1. Consider Table 1a to be a microdata table and the goal to create an anonymous version of that table satisfying 2-anonymity. Let attribute ID be the identifier, AGE and ZIP the QID with corresponding VGHs in Figure 1a and 1b. The SAs are given by the attributes DISEASE and TREATMENT. To provide an easy to follow example, we will not remove the ID. In the first iteration of Datafly the algorithm would pick attribute AGE for generalization, since it has the most different values: nine compared to eight of attribute ZIP. Every value is replaced by a more general value. Attribute ZIP still contains eight unique values, hence another generalization is performed on attribute ZIP. This table satisfies 1-anonymity, since the combination of ZIP and AGE is unique for tuple ID 5 (1236* and $[50-59]$. In order to satisfy 2-anonymity another iteration has to be performed. The generalization of attribute ZIP gives the resulting 2-anonymous table depicted in Table 1b, i. e., every individual in the table is indistinguishable from at least another individual in that table. The equivalence classes (ECs) are marked by dashed horizontal lines.

ID	ZIP	AGE	DISEASE	TREATMENT
1	12345	23	Gastric ulcer	Antacid
2	12345	29	Gastritis	Acid-reducing drug
3	12363	41	Flu	Antipyretic drug
4	12361	43	Stomach cancer	Cytostatic drug
5	12362	59	Pneumonia	Antibiotics
6	12471	52	Bronchitis	Antibiotics
7	12473	55	Flu	Antipyretic drug

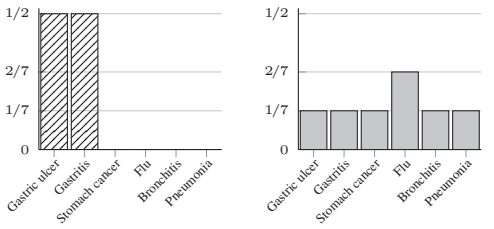
(a) Microdata-table

ID	ZIP	AGE	DISEASE	TREATMENT
1	123**	[20-29]	Gastric ulcer	Antacid
2	123**	[20-29]	Gastritis	Acid-reducing drug
3	123**	[40-49]	Flu	Antipyretic drug
4	123**	[40-49]	Stomach cancer	Cytostatic drug
5	123**	[50-59]	Pneumonia	Antibiotics
6	124**	[50-59]	Bronchitis	Antibiotics
7	124**	[50-59]	Flu	Antipyretic drug

(b) 2-anonymous table

Table 1: 2-anonymization of a microdata table

t-closeness The focus on the QIDs turned out to be a weak point of the concept of k-anonymity [MKGV07]. In [LLV07] Li et al. criticized the concept of k-anonymity for not taking the values of the SAs into account. Li et al. pointed out, that the semantic



(a) Frequency distribution of attribute DISEASE for the first EC (IDs {1,2})

(b) Frequency distribution of attribute DISEASE for the whole table

Figure 2: Comparison of frequency distribution

relationships between SA values of individuals may allow *attribute disclosure*. i. e., the association of an individual with the value of its SA or a *semantically close* value [LLV07]. To address the notion of closeness, Li et al. use the earth-movers distance (EMD) [RTG00].

The EMD is a metric used in image processing to measure the similarity of two pictures. The pictures are being described by the frequency distribution of colors occurring in the picture. Intuitively the EMD measures the amount of work necessary to transform one distribution into the other. The result is a value in the interval of $[0, 1]$. The closer the value is to 0 the closer the two distributions are.

Adapted to the use case of data anonymization, the similarity of attribute values is measured by comparing the frequency distribution of attribute values for every EC to the frequency distribution of the whole table. If the result is closer to 1, the values in an EC are very similar, thus allowing to infer information [LLV07]. Hence, a value closer to 0 is desirable. An example of the frequency distribution for the first EC (IDs {1,2}) for attribute DISEASE of Table 1b to the whole Table 1b is depicted in Figure 2a and Figure 2b. Based on Table 1b and a simplified VGH of attribute DISEASE in [LLV07] the table satisfies 0.57-closeness w. r. t. DISEASE[1]. If privacy requirements demand a lower t-closeness value, the table could be generalized once again using the Datafly-algorithm. The resulting table is a maximally generalized table containing only values of the root nodes of the VGHs. It satisfies 0-closeness, since it consists of only one EC.

2.2 Distributed data-anonymization

As outlined in section 1, today's interconnected society places new demands on data privacy. The challenge is to adapt to a computing environment where data is distributed across multiple distinct sites. Throughout this paper, we focus on physically distributed data *partitioned* across two distinct sites, i. e., medical institutions. In such an environ-

[1]Details of this calculation are left out due to space considerations. The reader is referred to [LLV07] for an in depth explanation of t-closeness

ment, data can be vertically or horizontally partitioned. In the following we focus on vertically partitioned data, i. e., data about the same individuals stored at multiple sites.

One problem when jointly computing an anonymization on data split across multiple sites, is to define a *security model*. A security model describes demands imposed on participating sites and what can be learned from executing the *protocol*, i. e., the distributed data anonymization algorithm. A typical security model is that of an *honest-but-curious (HBC) adversary*, i. e., no site gains information about another sites input other than can be inferred from the own input and the result of the computation [Gol04]. In an HBC model, each site acts accordingly to the protocol, i. e., every site is truthful about its input and further steps of the protocol [Gol04]. We adhere to this security model for the remainder of this paper. Solutions to this model involve either the use of a *trusted third party (TTP)* or the utilization of a *secure multiparty computation (SMC)* protocol [Gol04].

Using a TTP may not be possible at all times. Therefore, the paradigm of SMC describes a set of protocols that try to simulate the use of a TTP [Gol04]. This is achieved by using cryptographic functions hiding the input of participating sites and revealing nothing but the result of the computation.

One way of protecting the input is the use of a *commutative encryption scheme* as described by Shamir et al. in [SuLMA80]. An encryption scheme uses a *cryptographic function* $E_k : M \rightarrow M$, that transforms a *plaintext* $m \in M$ into a *cyphertext* $m \in M$ using a *key* k. We omit the details of E. The important property of a commutative encryption scheme is the independence of the order of encryption, i. e., given the encryption function E_k, two keys $0, 1$ and a plaintext $m \in M$, the following commutativity property holds: $E_0\left(E_1\left(m\right)\right) = E_1\left(E_0\left(m\right)\right)$.

2.3 DPP$_2$GA: distributed k-anonymity

The *Distributed Privacy-Preserving two-Party Generic Anonymizer (DPP$_2$GA)* protocol by Jiang et al. [JC05] implements distributed k-anonymity on vertically partitioned data. Its goal is to create a *globally* k-anonymous data set from two *locally* k-anonymous data sets, stored at different sites. During execution of the protocol, k-anonymity of the locally stored data sets is preserved. This definition of distributed privacy is related to the definition of SMC, but differs in that it allows *some* degree of information gain.

Before discussing the protocol in more detail we will outline its key steps: (*1*) Ensure local k-anonymity, (*2*) exchange encrypted information about local ECs, (*3*) ensure global k-anonymity and (*4*) join local data sets on a global identifier. We will elaborate on the key steps by using an example.

Example 2. The prerequisite of the protocol is a vertically partitioned table, distributed across two sites. An example is given in Table 2a reusing the running example from Table 1a split into two tables. For simplicity, we omit values for the SAs. The subscript number close to the attribute name indicates that the attribute is either stored at site 0 or site 1. As before, dashed horizontal lines indicate ECs. Following this abstract example, each local table is 1-anonymous, i. e., at least one individual is unique w. r. t. its QID values. We

ID$_0$	ZIP$_0$
1	12345
2	12345
3	12363
4	12361
5	12362
6	12471
7	12473

ID$_1$	AGE$_1$
1	23
2	29
3	41
4	43
5	59
6	52
7	55

(a) Initial state

ID$_0$	ZIP$_0$
1	1234*
2	1234*
3	1236*
4	1236*
5	1236*
6	1247*
7	1247*

ID$_1$	AGE$_1$
1	[20-29]
2	[20-29]
3	[40-49]
4	[40-49]
5	[50-59]
6	[50-59]
7	[50-59]

(b) 1st iteration

ID$_0$	ZIP$_0$
1	123**
2	123**
3	123**
4	123**
5	123**
6	124**
7	124**

ID$_1$	AGE$_1$
1	[20-39]
2	[20-39]
3	[40-59]
4	[40-59]
5	[40-59]
6	[40-59]
7	[40-59]

(c) 2nd iteration

Table 2: Distributed 2-anonymity using DPP$_2$GA

will assume that the participating sites agreed on publishing a globally 2-anonymous table. Therefore, following step (*1*), every site first creates a locally 2-anonymous version of its data set. As Jiang et al. [JC05] noted, this can be achieved using any k-anonymization algorithm. For simplicity, we assume the use of the Datafly-algorithm.

After the first iteration of Datafly on the local data sets, we obtain different ECs on both sites. The global result is identical to the result outlined in example 1. Hence, we observe the ECs depicted in Table 2b. Next, we investigate step (*2*) of the protocol. This step can be divided into five parts: (*2.1*) send an encrypted version of the ECs to the other site, (*2.2*) receive an encrypted version of the ECs from the other site, (*2.3*) encrypt received EC with own key and send it back to the other site, (*2.4*) receive an encrypted version of the EC sent to the other site in part (*2.1*) and (*2.5*) compare the encrypted multisets. To satisfy the k-anonymity requirement the sites need to exchange information about the equivalent classes in an encrypted, thus privacy-preserving way. Using the commutative encryption function, introduced in subsection 2.2, the two sites exchange their multisets.

We will explain this step by continuing our previous example, focusing on the first EC. Each site is now in the possession of the following encrypted multisets: site 0 : $\{\{E_0(1), E_0(2)\}, ...\}$ and site 1 : $\{\{E_1(1), E_1(2)\}, ...\}$. These encrypted IDs are being sent to the other site (part (*2.1*)). Site 0 now owns an encrypted version of the ECs of site 1. Furthermore, site 1 now owns an encrypted version of the ECs of site 0 (part (*2.2*)). Due to the encryption, neither site is able to read the IDs, therefore the k-anonymity requirement is still intact. As outlined in part (*2.3*), every site encrypts the version just received by the other site, with its own key and sends it back to the corresponding site. Now, every site owns two versions of the encrypted ECs: First, one originally encrypted by themselves and returned encrypted yet again by the other site. Second, the version originally received from the other site and now encrypted by themselves (part (*2.4*)). Since, the commutativity property holds for the encryption function E, we are now able to compare the ECs without knowing the actual values of the IDs (part (*2.5*)). For that purpose, Jiang et al. proved, that a local k-anonymization is globally k-anonymous, under the following circumstances: (*1*) For every encrypted ID in each EC in one multiset, there exists another EC in the other multiset containing the same ID. (*2*) The cut of those two sets is of cardinality at least k. We will refer to this property as the *multiset-equality property*. Continuing our example, this is not the case for ID 5 as can be seen in Table 2b. Hence, the local k-anonymizations are not globally k-anonymous. Therefore, another iteration of the Datafly-algorithm has to be performed on the local data. Afterwards, the comparison

has to be repeated. This process continues until a valid global k-anonymization is found. An example for a valid global k-anonymization can be seen in Table 2c.

Finally, in step (*4*) of the protocol the k-anonymous partitions have to be joined on the ID in order to form a k-anonymous publication. This can be achieved by using a secure join, as proposed by Jiang et al. in [JC06]. We will omit an explanation of this part, referring to [JC06] for further details.

3 Distributed two-party t-closeness

In this section, we present our protocol D2P*t*, which extends the DPP_2GA data anonymization technique by the concept of t-closeness.

3.1 Preliminary assumptions

For the following considerations, we make the same assumptions as in subsection 2.3, i. e., the data is vertically partitioned across two independent sites. Furthermore, the sites agreed on the parameters of the anonymization protocol prior to executing the protocol, e. g., k-value for k-anonymity, t-value for t-closeness, a common ID describing the same individual on both sites, as well as the parameters used for the commutative encryption scheme. In addition we use the same security model used in subsection 2.3, e. g., an HBC adversary. Lastly, we assume the privacy model described in [JC05], thus allowing some inference of information, hence weakening the rules of SMC. Since protocols of SMC are costly and add an additional layer of complexity, we decided to lose the absolute security of SMC but gain an intuitive and easy to follow protocol. Also, methods of computer and communication security are out of scope of this work. As Sweeney emphasizes in [Swe02b], we state that every entry in the table be specific to one individual only.

3.2 Problem statement

To clarify our concept of t-closeness in a distributed environment, we have to discuss some important observations. As we have shown in subsection 2.3, the concept of k-anonymity can be transferred to a distributed environment. Enhancing this approach with the concept of t-closeness has two shortcomings. First, as was shown in section 2.1, the t-closeness property incorporates the notion of semantics of SA values to an anonymization. Since the SA is the attribute an anonymization is trying to protect, information about it cannot be exchanged directly. Therefore, only relying on the cut-operation as an indicator of a globally valid anonymization is not enough. The cut may shrink an EC as long as its cardinality is still greater or equal to k. A reduced EC has a drastic influence on the t-closeness property, thus shrinking an EC is not an option when using t-closeness. Secondly, a joined data publication imposes several problems. On the one hand side, it is

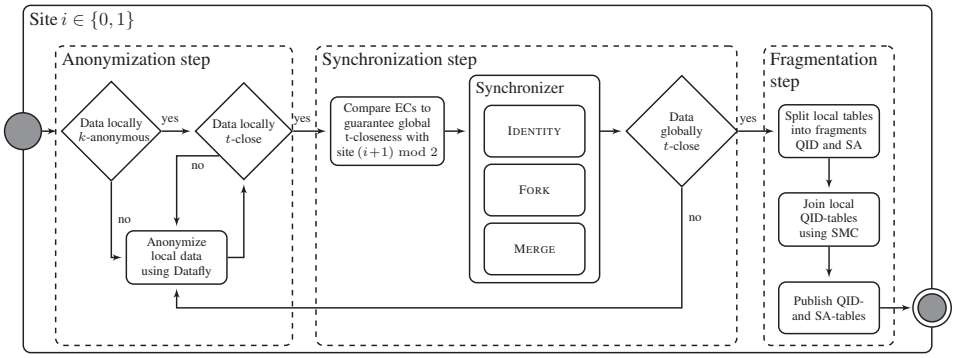


Figure 3: Flowchart of the protocol D2P*t*

desirable to publish the combination of every SA for each individual. On the other hand, as Fang et al. [FAN11] pointed out, this changes the characteristics of the SA. Since the original values of each SA are known to the site publishing it, this site could re-identify the value in the anonymized data set, thus re-identifying an individual. Therefore, the SA values transform into additional QIDs, which we term *identifying sensitive attributes (iSAs)*. iSAs introduce a new threat to k-anonymity. Fang et al. [FAN11] identified this threat as the *background-join attack*. Lastly, it remains unclear how data privacy models like t-closeness, that operate on frequency distributions of SA values handle the case of multiple sensitive attributes (MSA). Computing t-closeness on the joint distribution of the SAs does not provide a feasible solution, since it will most likely be evenly distributed, thus providing no useful information.

Our observations lead to the following requirements for a global t-closeness in a distributed two-party environment having multiple sensitive attributes: (*1*) The released data must be t-close for every SA. (*2*) The k-anonymity-property must respect the background-join attack based on iSAs.

3.3 Overview

A schematic overview of our protocol D2P*t* can be found in Figure 3. It depicts an instantiation of the key components at one site. The protocol consists of three components: (*1*) the *Anonymizer*, (*2*) the *Synchronizer* and (*3*) the *Fragmenter*.

3.4 Anonymization phase: k-anonymity and t-closeness

The Anonymizer creates a locally k-anonymous data set, using an anonymization algorithm. For the sake of simplicity, we chose the Datafly-algorithm which was introduced in section 2.1. Although we are not limited to the Datafly-algorithm, as any algorithm

ID_0	ZIP_0	ID_1	AGE_1
1	1234*	1	[20-29]
2	1234*	2	[20-29]
3	1236*	3	[40-49]
4	1236*	4	[40-49]
5	1236*	5	[50-59]
6	1247*	6	[50-59]
7	1247*	7	[50-59]

(a) 1st iteration

ID_0	ZIP_0	ID_1	AGE_1
1	123**	1	[20-39]
2	123**	2	[20-39]
3	123**	3	[40-59]
4	123**	4	[40-59]
5	123**	5	[40-59]
6	124**	6	[40-59]
7	124**	7	[40-59]

(b) 2nd iteration

ID_0	ZIP_0	ID_1	AGE_1
1	*	1	*
2	*	2	*
3	*	3	*
4	*	4	*
5	*	5	*
6	*	6	*
7	*	7	*

(c) 3rd iteration

Table 3: Distributed t-closeness using D2Pt-IDENTITY

guaranteeing full-domain generalization is applicable to provide the anonymization phase. The k-anonymous data is handed over to the t-closeness-verifier, that in turn verifies the t-closeness of the data for a given t. If that check fails, the data is being handed back to the Anonymizer to produce a more general version of the anonymized table. After a finite number of iterations the table satisfies t-closeness. A table will satisfy t-closeness after a finite number of runs, since any generalization algorithm will produce a maximally generalized table at some point of its execution. The maximally generalized table is equal to the whole table thus satisfying 0-closeness, as outlined in section 2.1.

3.5 Synchronizer

The Synchronizer component handles the comparison of the two local anonymizations and decides whether they fulfill the needs of a globally t-close data set or not. It uses the mechanism of commutative encryption outlined in subsection 2.2 and subsection 2.3 to compare the anonymized tables produced by the two sites. In case of a negative result the local anonymization is handed back to the Anonymizer to perform another iteration of the anonymization algorithm. We developed three versions of the Synchronizer. Each one providing special properties to the protocol, as will be outlined below.

Synchronizer: IDENTITY The IDENTITY Synchronizer is the simplest form of a Synchronizer we developed. Its mode of operation is a straight-forward adaption of the synchronization protocol used by Jiang et al. [JC05]. Instead of demanding a cut of cardinality of at least k, each EC has to be identical on both sites.

Example 3. Revisiting our running example, Table 3a displays the state of the Synchronizer after the first iteration of Datafly. Again, we omit the SA values for clarity of the example. The dashed horizontal lines depict ECs. Due to ID 5 no equal ECs exist, thus the data is being handed back to the Anonymizer to create a more general anonymization. The result of the next synchronization attempt is shown in Table 3b. Once again the ECs do not match, hence demanding another generalization. This step is the final anonymization, since its result is a maximally generalized table, as can be seen in Table 3c. Therefore, the synchronization process is complete and the data is being transferred to the Fragmenter.

ID_0	ZIP_0
1	1234*
2	1234*
3	1236*
4	1236*
5	1236*
6	1247*
7	1247*

ID_1	AGE_1
1	[20-29]
2	[20-29]
3	[40-49]
4	[40-49]
5	[50-59]
6	[50-59]
7	[50-59]

(a) 1st iteration

ID_0	ZIP_0
1*	1234*
2*	1234*
3	123**
4	123**
5	123**
6	124**
7	124**

ID_1	AGE_1
1*	[20-29]
2*	[20-29]
3	[40-59]
4	[40-59]
5	[40-59]
6	[40-59]
7	[40-59]

(b) 2nd iteration

ID_0	ZIP_0
1*	1234*
2*	1234*
3	*
4	*
5	*
6	*
7	*

ID_1	AGE_1
1*	[20-29]
2*	[20-29]
3	*
4	*
5	*
6	*
7	*

(c) 3rd iteration

Table 4: Distributed t-closeness using D2Pt-FORK

Synchronizer: FORK The FORK Synchronizer is an improvement of the IDENTITY Synchronizer. We observed that, even though identical ECs exist, they are lost during the next generalization phase. The essential part of the FORK Synchronizer is its ability to mark identical ECs and exclude them from the next generalization phase.

This marking is also conducted in a privacy-preserving way. Therefore, every EC is assigned a separate ID, the *ec-ID*, which is also being encrypted using a commutative encryption scheme. After recognizing an identical EC, the FORK Synchronizer decrypts its ec-ID and sends it to the other site. In turn it receives an decrypted version of the ec-ID of the other site. Thus allowing both sites to decrypt the ec-ID and gain access to the information what EC to exclude from the next generalization phase.

Example 4. As an example review Table 4a derived from the initial Table 1a. The first EC containing the IDs 1 and 2 is equal on both sites. Thus the FORK-Synchronizer marks them for exclusion, depicted by an asterisk after the ID. The next iteration excludes the tuples from the first EC. The result is shown in Table 4b. Note, that no additional matching ECs are being found. Hence, a third iteration yields the result depicted in Table 4c.

Synchronizer: MERGE Revisiting the previous example, the following observation can be made. The IDs $\{3, 4, 5, 6, 7\}$ seem to be candidates for alignment, as can be seen in Table 5a. Unfortunately, the next generalization phase creates different ECs which render such a generalization impossible. The MERGE Synchronizer addresses this issue. We explain its behavior given the following example.

Example 5. The MERGE Synchronizer randomly selects one EC and adds its IDs to a queue Q which cannot contain duplicates. Let $\{6, 7\}$ be that EC, hence the queue consists of $Q = (6, 7)$. For each element in the queue on both sites the EC containing this element is being identified. The EC is then marked for not being available anymore, analogously to the FORK Synchronizer. We will select element 6. Every element in that EC is added to the queue, therefore, the elements $(5, 6, 7)$ from site 1 are added to the queue, e. g., $Q = (5, 6, 7)$. In the next step MERGE selects ID 5, leading to the addition of elements $(3, 4, 5)$ from site 0. The resulting queue is $Q = (3, 4, 5, 6, 7)$. After that no other elements are being added to the queue, indicating that MERGE found a valid merge of ECs, which, when generalized accordingly, will form a valid anonymization.

ID_0	ZIP_0
1	1234*
2	1234*
3	1236*
4	1236*
5	1236*
6	1247*
7	1247*

ID_1	AGE_1
1	[20-29]
2	[20-29]
3	[40-49]
4	[40-49]
5	[50-59]
6	[50-59]
7	[50-59]

(a) 1st iteration

ID_0	ZIP_0
1	1234*
2	1234*
3	*
4	*
5	*
6	*
7	*

ID_1	AGE_1
1	[20-29]
2	[20-29]
3	[40-59]
4	[40-59]
5	[40-59]
6	[40-59]
7	[40-59]

(b) 2nd iteration

GID	ZIP_0	AGE_1
1	1234*	[20-29]
1	1234*	[20-29]
2	*	[40-59]
2	*	[40-59]
2	*	[40-59]
2	*	[40-59]
2	*	[40-59]

GID	DISEASE
1	Gastric ulcer
1	Gastritis
2	Flu
2	Stomach cancer
2	Pneumonia
2	Bronchitis
2	Flu

GID	TREATMENT
1	Antacid
1	Acid-reducing drug
2	Antipyretic drug
2	Cytostatic drug
2	Antibiotics
2	Antibiotics
2	Antipyretic drug

(c) Resulting data publication

Table 5: Distributed t-closeness using D2Pt-MERGE

3.6 Fragmenter

Finally, the Fragmenter component is responsible for joining and releasing the data set. As its name suggests the resulting tables are not being joined and published as a whole. Rather, the data handling is twofold. On the one hand side, the quasi identifiers of sites 0 and 1 are being joined on the ID using a secure join protocol proposed by Jiang et al. [JC06]. On the other hand, the data is being partitioned, i. e., QIDs and SAs are being split into different tables. To maintain a reference from the QIDs to the SAs a new attribute, termed *global identifier (GID)* is inserted. The GID associates one SA value to at least k QID values, depending on the size of the EC. After the fragmentation the QIDs can be restored to their original values, because no direct association between a QID value and a SA value exists [XT06]. This process eliminates the risk for a background-join attack based on iSAs and, allow the t-closeness to be applied on each SA individually.

3.7 Privacy analysis

Privacy preservation We analyze our distributed anonymization protocol D2Pt w. r. t. compliance to k-anonymity and t-closeness. We show the validity of our approach using the following theorems with sketches of proofs.

The proof is split into three parts. First, we show compliance with k-anonymity during the execution of the protocol. Second, we extend this property to the t-closeness property. Lastly, we show that the fragmented result set adheres to the k-anonymity and t-closeness property.

Theorem 1. *The Synchronizer component introduces no risk of invalidating the k-anonymity property of the local data.*

Proof sketch. We need to show that the Synchronization step shown in Figure 3 does not break the k-anonymity properties of the local anonymization. First, we outline the basic setup and then go into detail about the k-anonymity property and discuss the t-closeness property thereafter.

The Synchronizer receives a k-anonymous and t-close data set from the Anonymizer component. The anonymization is valid w. r. t. to the local data only. Now, exchanging information about the ECs in an encrypted, thus privacy preserving way does not invalidate the k-anonymity property of the data. As outlined in subsection 3.2, the Synchronizer component demands the ECs to be identical in terms of the contained IDs. This is a special case of the multiset-equality property introduced in subsection 2.3. In case the synchronization step fails, the data is being handed back to the Anonymizer. The Anonymizer produces a more general version of the data, which will lead to larger ECs containing the same set of tuples as in the previous iteration plus additional tuples from another EC.

Theorem 2. *The Synchronizer component introduces no risk of invalidating the t-closeness property of the local data.*

Proof sketch. Following the idea of theorem 1, the Synchronizer receives a more general version of the local data in every iteration of the protocol. Due to the generalization property, outlined by Li et al. in [LLV07], the t-closeness is invariant to further generalizations. By using commutative encryption and exchanging information only on the IDs, neither site is able to learn anything about the frequency distribution of the SA-values. Following the same argument as Jiang et al. in [JC05], the data exchange step does not reveal information about the t-closeness of the local data.

Theorem 3. *The distributed anonymization protocol D2Pt computes a k-anonymous and t-close view of vertically partitioned data in an HBC model.*

Proof sketch. To show the compliance of the proposed protocol with the k-anonymity and t-closeness property, we have to analyze the result sets. First, we will elaborate on the k-anonymity property of the global data, e. g., the data depicted in Table 5c. As explained in subsection 3.6, each tuple contains a reference to the SA-tables, e. g., the GID. Since the GID is derived using the ECs constructed by the Datafly-algorithm, at least k tuples are referencing the same set of at least k SA-values. Following the works of Xiao et al. such fragmented data fulfills the k-anonymity property [XT06].

Using the fragmentation, we also maintain the t-closeness property on the local data throughout the release of the global data set. We will elaborate on this proposition by outlying a sketch of the steps necessary to prove this statement.

We start by reviewing Table 5c. As a result of the global anonymization process, we obtain three tables, e. g., a QID-table and two SA-tables. We will refer to the table containing the SA DISEASE as SA_0 and the table containing the SA TREATMENT as SA_1. We need to show the validity of the t-closeness for every SA in the set resulting from our protocol.

The two tables SA_0 and SA_1 contain a set of identical ECs. That is, the GID attribute references an identical set of individuals on both sites. Using the GID is the only way to

reconstruct the original data. Therefore, the tables SA_0 and SA_1 can only be joined on their GID. Joining the tables on the GID results in a table $SA_\times$, by taking the Cartesian product of every EC from SA_0 with the matching EC of SA_1.

Given $SA_\times$, it is not useful to derive the t-closeness for SA_0 or SA_1. As the set includes a large number of tuples not included in the original data, called *phantom tuples* hereafter, the calculation of the t-closeness provides no information gain w. r. t. the original data. The amount of phantom tuples increases in the cardinality of the ECs of $SA_\times$. To estimate the amount of phantom tuples, we assume a minimal value for k-anonymity of 2. Thus, every EC introduces at least one phantom tuple for each existing tuple in the tables SA_0 and SA_1. In order to utilize the released data $SA_\times$ to infer information on the t-closeness, $SA_\times$ has to be modified. Specifically, the phantom tuples have to be removed, since they introduce perturbation of the data. This can be achieved based on the frequency distribution of SA_0 and SA_1. The goal is to reconstruct a set of tuples, termed $SA'_\times$, that includes one tuple for each individual in the original data.

We show the reduction of $SA_\times$ by revisiting our running example from Table 5c, using only the first EC with GID 1. We observe, that attribute value "Gastritis" of attribute DISEASE, would appear twice in $SA_\times$, e. g., "(Gastritis, Acid-reducing drug)" and "(Gastritis, Antacid)". One of these tuples must be a phantom tuple, since Table 5c contains "Gastritis" only once. Without background knowledge, we can only select one combination by chance. Next, we extend this example to the use of background knowledge.

Given the knowledge that Acid-reducing drugs are a promising treatment for Gastritis, we would select the corresponding tuple "(Gastritis, Acid-reducing drugs)" as candidate for the reconstruction. Therefore, we remove the identified phantom tuple "(Gastritis, Antacid)" from $SA_\times$. Applying this procedure to every EC results in $SA'_\times$, which contains every sensitive attribute value that appears in SA_0 or SA_1. It has to be noted, that the resulting tuples will not necessarily represent an individual present in the original table. Most important, the number of tuples in $SA'_\times$ equals the number of tuples in SA_0 or SA_1. Additionally, the frequency of every value in SA_0 or SA_1 is exactly the same as in $SA'_\times$. Thus, the frequency distribution of the attribute values remains unchanged. Since t-closeness is computed from the frequency distribution of attribute values, its result is the same compared to the t-closeness gained by the Synchronizer.

To sum up our proof sketch, we have shown that: (*1*) k-anonymity is maintained during the execution of the protocol in theorem 1, (*2*) t-closeness is maintained during the execution of the protocol in theorem 2 and (*3*) the result does adhere to the k-anonymity and t-closeness property, as shown in theorem 3.

As additional findings, we want to elaborate on properties regarding the calculation of the t-closeness for table $SA_\times$. First, joining the table SA_0 and SA_1 on the GID resulting in the table $SA_\times$. If the table $SA_\times$ contains ECs different in size, the resulting t-closeness computed over the frequency distribution of every attribute, will be different from the one computed on the local data sets. The bigger the difference in size between the smallest and largest EC, the more the t-closeness differs. Second, if the data in the table $SA_\times$ is skewed or correlated it is possible to infer information about the original distribution from the frequency distribution in the table $SA_\times$. For example, one could use the conditional

probability measure to find the most likely join candidate among the tuple combinations in every EC. This allows the removal of impossible combinations in the whole table, e. g., the attribute value appears only once. Furthermore, our experiments indicate that it will likely be possible to reconstruct the original table, thus linking the SA-values. In subsection 3.6 this was identified as being a privacy threat. If the SA-values are unique a re-identification has appeared. However, the t-closeness is a privacy metric that does not address this kind of attack and thus cannot protect from it. The reconstructed table still adheres to t-closeness.

4 Experimental Evaluation

In this section, we evaluate our protocol D2P*t* presented in section 3. At first, we describe our experimental setup in subsection 4.1. Then, we compare the results with respect to the Synchronizer in section 4.2, i. e., the amount of synchronization attempts. We have also conducted an in-depth experimental analysis of the amount of data transmitted. We will leave out a detailed presentation of the results due to the focus of the paper being on the concepts of data anonymization and their adaption to a distributed environment. Finally, we analyze data utility of the published data in section 4.2.

4.1 Experimental setup

Prototypical implementation We implement D2P*t* as a prototype in Java. The prototype consists of a main program and two logical units representing the different sites. Each site holds an instance of an Anonymizer responsible for ensuring the k-anonymity and t-closeness properties. The Datafly-implementation is based on the Anonymization Toolbox of UT Dallas[2]. In contrast to the Anonymization Toolkit, which uses SQLite as a data store, we chose PostgreSQL to allow concurrent access to the database.

In a preparation step, we introduced a cache structure, thus, allowing us to reuse anonymization results from the Datafly-algorithm, therefore, significantly accelerating the process of data anonymization. In this paper, we exclude the preparation step from our measurements, since times needed for data anonymization and exchange are not in the focus of our evaluation. As mentioned in section 3 the anonymization algorithm is exchangeable, thus providing no use for a comparison of the anonymization algorithm. Rather, we focus on the simplicity and comprehensibility of our protocol. Since we are only interested in the utility of the anonymization results and the amount of data transmitted during the synchronization phase, we also exclude an evaluation of the commutative encryption scheme.

Data description We evaluate our protocol using the Adult Data Set[3], which is the de facto standard for benchmarking anonymization techniques [FTH+11]. Out of the approx-

[2] `http://cs.utdallas.edu/dspl/cgi-bin/toolbox/index.php`
[3] `https://archive.ics.uci.edu/ml/datasets/Adult`

imately $50\,000$ entries forming the training data, we use only those containing no NULL values, i. e., $30\,162$ entries. As outlined in section 2.1 and section 2.1, we need a set of VGHs to model possible generalizations, as well as semantic relationship between attribute values. We adapt the VGHs presented by Fung et al. in [FWY05]. We classify the attributes contained in the Adult Data Set into QIDs and SAs. The QIDs are formed using the attributes *Workclass, Education, Marital Status, Relationship, Race, Sex* and *Native Country*. Likewise, the SAs are formed using the attributes *Occupation* and *Salary*. This is equivalent to the setting used by Li et al. [LLV07].

Workload For our test series we define the following schedule: We randomly assign each site one SA. Afterward, the QIDs attributes are distributed among the sites in every possible combination. Every combination of QIDs is evaluated with every combination of k, t values and every Synchronizer. For values of k we adapt the ones used by Jiang et al. in [JC05], namely, $k \in \{2, 5, 10, 20, 100\}$. Likewise, we use the values given for t by Li et al. [LLV07], i. e., $t \in \{0.1, 0.2\}$. To get a better understanding of the influence of high t values we add the value 0.7 and 1.0. The latter, allows us to simulate DPP_2GA using a specialized Synchronizer, without the need to modify out protocol. Thus we obtain $t \in \{0.15, 0.2, 0.7, 1.0\}$

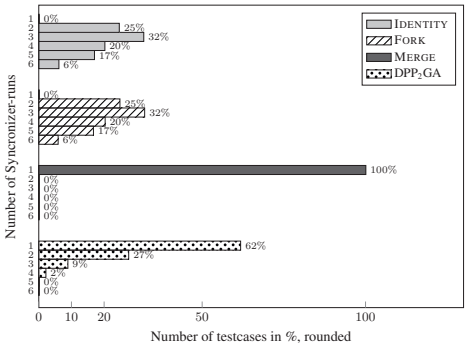
4.2 Experimental results

In the previous section we described the experimental setup. We will now analyze the impact of the k and t values on the resulting anonymization. Therefore, we differentiate our results into results called SAME and DIFF. Results stemming from SAME are obtained from initially identical anonymizations produced by the Anonymizer, i. e., the same ECs exist at both sites after the first anonymization phase. Results obtained from initially different anonymizations, will be called DIFF. The reason for the distinction between the two is given in Figure 4a. We observe that, given a t value of 0.15, the amount of initially identical anonymizations is almost $30\,\%$ for every k. That can be explained by the reduced amount of possible generalizations available for the strict privacy constraints of a small t. An initially identical anonymization renders the synchronization phase useless, since it evaluates to true after the first iteration. Since we want to observe the impact of t on the anonymization process at a whole, we distinguish between the cases SAME and DIFF. The following numbers apply to numbers obtained by DIFF unless otherwise stated. The term ALL will refer to the combined results of SAME and DIFF.

Synchronization attempts As outlined in subsection 3.5, situations occur when every Synchronizer except MERGE has to perform more than one generalization step. Since this is only the case for ECs stemming from DIFF, we directly observe the need for such a distinction. As expected, the MERGE-Synchronizer always performs its synchronization within one step. DPP_2GA also performs well during the synchronization phase, i. e., it needs more than three attempts in less than $2.1\,\%$ of all testcases. In contrast, the Syn-

t	k = 2	5	10	20	50	100
0.15	29.4%	29.4%	29.4%	30.1%	30.1%	30.1%
0.2	5.5%	5.5%	5.5%	6.0%	6.0%	6.0%
0.7	1.2%	2.2%	2.5%	2.7%	3.2%	3.5%
1	1.6%	2.2%	2.5%	2.7%	3.2%	3.5%

(a) Percentage of SAME-share w. r. t. k and t



(b) Distribution of Syncronizer-runs

Testcase	Synchronizer	t = 0.15	0.2	0.7	1.0
DIFF	IDENTITY	1.00	1.00	0.97	0.97
	FORK	1.00	0.99	0.96	0.95
	MERGE	0.99	0.84	0.63	0.63
	DPP_2GA	-	-	-	0.22
SAME	IDENTITY	0.60	0.58	0.32	0.32
	FORK	0.60	0.58	0.32	0.32
	MERGE	0.60	0.58	0.32	0.32
	DPP_2GA	-	-	-	0.32
ALL	IDENTITY	0.88	0.97	0.95	0.95
	FORK	0.88	0.97	0.94	0.94
	MERGE	0.87	0.82	0.63	0.62
	DPP_2GA	-	-	-	0.22

(c) Normalized discernibility metric dependent on the Synchronizer and t

Figure 4: Evaluation of D2P*t*

chronizers IDENTITY and FORK need more than three synchronization attempts in over $40\,\%$ of the cases. The only situations where IDENTITY and FORK need less than three Synchronizer runs, is given by small t values, e. g., $\{0.15, 0.2\}$. That can be explained by the reduced amount of possible generalizations available for the strict privacy constraint of a small t.

Data utility As a measure to rate the utility of an anonymized data set we use the *discernibility metric (DM)*. This metric was introduced by Bayardo et al. [BA05]. It penalizes large ECs. Its result is the sum of these penalties. In order to gain comparable results, the sum has to be normalized. The evaluation on the utility dependent on the Synchronizer and t value is shown in Figure 4c.

As a pre-note, we want to emphasize, that results for DPP_2GA only apply for the case of $t = 1$, since the protocol does not support t-closeness. The results obtained by the MERGE-Synchronizer are generally preferable to those of IDENTITY and FORK. Surprisingly, FORK does not perform much better than IDENTITY. While it is able to mark ECs in $61\,\%$ of its testcases, this only slightly affects the result of the DM. Not surprisingly, all Synchronizers show their worst performance for small t values. As noted before, a small t leads to maximal generalization with high probability.

Again, we can observe the use of splitting the result into the groups of SAME and DIFF. Clearly, every Synchronizer in the SAME group obtains the same utility, since no synchronization has to be performed after the first anonymization phase.

In summary, our results indicate that the utility of the data generated by D2Pt is lower than that of DPP$_2$GA. However, this also results from the use of t-closeness as a privacy metric and its higher demands on the structure of ECs.

5 Related work

The protocol DPP$_2$GA has been criticized for relaxing the notion of SMC. Therefore, Jiang et al. [JC06] extended DPP$_2$GA to the protocol DkA to comply with the rules of SMC. Both protocols perform a *bottom-up generalization* by gradually generalizing QID values until achieving k-anonymity.

Mohammed et al. [MFD11] propose a *top-down* variant of a distributed k-anonymization protocol. This protocol, iteratively specializes maximally generalized QID values, thus producing an approximate optimal k-anonymity.

Recent work by Kohlmayer et al. [KPEK13] presents a study on state-of-the-art anonymization algorithms applied to a distributed environment [KPEK13]. The proposed framework is capable of dealing with horizontally and vertically partitioned data sets. One of the concepts discussed by Kohlmayer et al. is the concept of t-closeness. However, as Kohlmayer et al. pointed out, their framework is not capable of dealing with numerical values in case of t-closeness. Furthermore, it was not shown if the framework respects the threats imposed by the background-join attack introduced by Fang et al. [FAN11].

Closely related to the adaption of privacy metrics to a distributed environment is the question of dealing with multiple sensitive attributes (MSA). This setting is imposed by a distributed scenario where each participating institution holds at least one SA.

The problem of MSA does not affect k-anonymization algorithms, since k-anonymity does not require the SAs to satisfy any properties. The concept of ℓ-diversity demands SAs values to be "well-represented" [MKGV07]. Machanavajjhala et al. identify the problem of MSA, but omit a detailed discussion. Recent work by Das et al. [DB12] addresses the problem of MSA and provide a solution specifically tailored for ℓ-diversity. The solution of Das et al., *Decomposition+*, is related to ours, since it also fragments the data.

Data fragmentation has been discussed before in the context of k-anonymization. It is achieved by the *Anatomy*-algorithm developed by Xiao et al. [XT06]. Recently, Fang et al. [FAN11] addressed the problem of MSA for centralized t-closeness. Providing a fragmenting solution, that analyzes the information loss imposed by the fragmentation.

6 Conclusion & future work

Throughout this paper, we have shown a simple yet effective method to enhance the distributed k-anonymization algorithm DPP$_2$GA with the privacy requirements of t-closeness. Using the MERGE-Synchronizer, we were able to significantly enhance the utility of our distributed anonymization protocol.

Moreover, the results we have gathered so far, indicate that the globally t-close table is identical to the one produced by a centralized instance of the Datafly-algorithm. However, a more detailed discussion on this assumption is subject to future research.

Nevertheless, our evaluation shows that DPP_2GA achieved better utility. Therefore, it should be noted, that the concept of t-closeness itself lowers utility of the data. In fact, Li et al. criticized that it "substantially limits the amount of useful information that can be extracted from the released data" [LLV10].

The work presented in this paper focuses on the simplicity of the protocol, thus relaxing some security requirements typically imposed by the notion of SMC. Future work could focus on allowing distributed t-closeness under the strict requirements of SMC. Furthermore, our current protocol is limited to a two-party design, thus allowing only two sites to jointly compute an anonymization. It would be a challenging task to adopt our protocol to an n-party environment, i. e., an arbitrary number of participating sites. Finally, it would be of great interest to investigate the benefits of bottom-up and top-down approaches and their application in an distributed environment.

References

[AFK+05] G. Aggarwal, T. Feder, K. Kenthapadi, R. Motwani, R. Panigrahy, D. Thomas, and A. Zhu. Approximation Algorithms for k-Anonymity. *Journal of Privacy Technology*, 2005.

[BA05] R. J. Bayardo and R. Agrawal. Data Privacy through Optimal k-Anonymization. In *Proceedings of the 21st International Conference on Data Engineering*, 2005.

[CDCdVFS07] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Microdata protection. In *Secure Data Management in Decentralized Systems*, 2007.

[CT13] C. W. Clifton and T. Tassa. On syntactic anonymity and differential privacy. In *ICDE Workshop on Privacy-Preserving Data Publication and Analysis*, 2013.

[Dal77] T. Dalenius. Towards a methodology for statistical disclosure control. *Statistisk Tidskrift*, 15(15):429–444, 1977.

[DB12] D. Das and D. K. Bhattacharyya. Decomposition+: Improving ℓ-Diversity for Multiple Sensitive Attributes. In *Advances in Computer Science and Information Technology. Computer Science and Engineering*, pages 403–412. 2012.

[FAN11] Y. Fang, M. Ashrafi, and S. Ng. Privacy beyond Single Sensitive Attribute. In *Database and Expert Systems Applications*. 2011.

[FED94] Federal Committee on Statistical Methodology. *Report on Statistical Disclosure Limitation Methodology*, 1994.

[FTH+11] B. C. M. Fung, T. Trojer, P. C. K. Hung, L. Xiong, Khalil A.-H., and R. Dssouli. Service-Oriented Architecture for High-Dimensional Private Data Mashup. *IEEE Transactions on Services Computing*, 2011.

[FWY05] B. C. M. Fung, K. Wang, and P. S. Yu. Top-down specialization for information and privacy preservation. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 205–216, 2005.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, 1 edition, 07 2004.

[JC05] W. Jiang and C. W. Clifton. Privacy-Preserving Distributed k-Anonymity. In *Working Conference on Data and Applications Security*, 2005.

[JC06] Wei Jiang and Christopher W. Clifton. A secure distributed framework for achieving k-anonymity. *The VLDB Journal*, 15:316–333, 11 2006.

[KPEK13] F. Kohlmayer, F. Prasser, C. Eckert, and K. A. Kuhn. A flexible approach to distributed data anonymization. *Journal of Biomedical Informatics*, 2013.

[LeF07] K. LeFevre. *Anonymity In Data Publishing And Distribution*. PhD thesis, 2007.

[LLV07] N. Li, T. Li, and S. Venkatasubramanian. t-Closeness: Privacy Beyond k-Anonymity and ℓ-Diversity. In *Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[LLV10] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. Closeness: A New Privacy Measure for Data Publishing. *IEEE Transactions on Knowledge and Data Engineering*, 22:943–956, 2010.

[MFD11] Noman Mohammed, Benjamin C. M. Fung, and Mourad Debbabi. Anonymity meets game theory: secure data integration with malicious participants. *The VLDB Journal*, 20:567–588, 2011.

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data*, 1, 03 2007.

[MW04] Adam Meyerson and Ryan Williams. On the complexity of optimal K-anonymity. In Catriel Beeri and Alin Deutsch, editors, *PODS*, pages 223–228. ACM, 2004.

[RTG00] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 11 2000.

[Sam01] Pierangela Samarati. Protecting Respondents' Identities in Microdata Release. *IEEE Transactions on Knowledge and Data Engineering*, 13:1010–1027, 2001.

[SuLMA80] Adi Shamir and Ronald L. Rivest und Leonard M. Adleman. *Mental poker*, pages 37–43. Springer US, 1980.

[Swe97] Latanya Sweeney. Guaranteeing anonymity when sharing medical data, the Datafly System. *Proceedings : a conference of the American Medical Informatics Association*, pages 51–55, 10 1997.

[Swe02a] Latanya Sweeney. Achieving k-Anonymity Privacy Protection Using Generalization and Suppression. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):571–588, 10 2002.

[Swe02b] Latanya Sweeney. k-Anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.

[US96] 104th Congress. *Health insurance portability and accountability act*, 1996.

[XT06] Xiaokui Xiao and Yufei Tao. Anatomy: Simple and Effective Privacy Preservation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 139–150. ACM, 2006.

Admission Control für kontinuierliche Anfragen

Timo Michelsen, Marco Grawunder, Cornelius Ludmann, H.-Jürgen Appelrath

Universität Oldenburg, Department für Informatik
Escherweg 2, 26129 Oldenburg
{timo.michelsen, marco.grawunder, cornelius.ludmann, appelrath}@uni-ol.de

Abstract: Überlast bedeutet, dass an ein System mehr Anforderungen gestellt werden, als es erfüllen kann. Im schlechtesten Fall ist es nicht mehr ansprechbar oder stürzt ab. Für Datenbankmanagementsysteme (DBMS) existiert eine spezielle Komponente, Admission Control (AC) genannt, welche die Systemlast überwacht, eintreffende Anfragen vor deren Ausführung überprüft und ggfs. zurückstellt. Für kontinuierliche Anfragen, welche permanent ausgeführt werden, reicht diese Art von AC jedoch nicht aus: Es kann zur Konfliktlösung nicht mehr auf die Terminierung einer Anfrage gewartet werden. Zudem ist die Verarbeitung datengetrieben, d. h. Umfang und Inhalt der Daten können stark variieren.

Diese Arbeit stellt ein Konzept vor, wie eine flexible und anpassbare AC-Komponente für kontinuierliche Anfragen auf Basis eines Event-Condition-Action-Modells (ECA) umgesetzt werden kann. Zur Regeldefinition wird die einfache Sprache CADL vorgestellt, die eine hohe Flexibilität und Anpassbarkeit der kontinuierlichen AC an konkrete Systeme und Hardware erlaubt. Die Evaluation zeigt mittels Odysseus, einem Framework für Datenstrommanagementsysteme, dass das Konzept effizient funktioniert und effektiv zur Lastkontrolle und Ergreifung von Maßnahmen zur Lastreduktion eingesetzt werden kann.

1 Einleitung

Für Datenbanksysteme ist es essentiell, jederzeit eine reibungslose Verarbeitung der Daten sicherzustellen. Überlastungen können Antwortzeiten vergrößern, wirtschaftliche Kosten verursachen und im Extremfall zu Ausfällen führen. Solche Situationen können entstehen, wenn das Datenbanksystem eine große Menge an Anfragen verarbeiten muss. Daher ist es wichtig, Überlastsituationen generell zu vermeiden oder schnell aufzulösen. Dementsprechend müssen Datenbanksysteme in der Lage sein, ihre aktuelle Systemauslastung zu überwachen und in kritischen Situationen Maßnahmen zu ergreifen. Dieser Vorgang der Lastüberwachung und -kontrolle wird i. Allg. als *Admission Control* (AC) bezeichnet.

Für klassische Datenbankmanagementsysteme (DBMS) existieren bereits ausgereifte AC-Konzepte [HSH07]. Ist die vermutete zusätzliche Systemlast einer neuen Anfrage zu hoch (z. B. fehlender freier Arbeitsspeicher), kann die Anfrage abgelehnt, zurückgestellt, ausgelagert oder nur unter eingeschränkten Rahmenbedingungen ausgeführt werden. Diese Möglichkeiten sind für einmalig ausgeführte Anfragen ausreichend. Es existieren jedoch Anwendungsszenarien, die eine kontinuierliche und zeitnahe Verarbeitung erfordern. Dort

senden *aktive Datenquellen* autonom theoretisch unbegrenzte Mengen an Daten, sogenannte *Datenströme* [Krä09]. Diese können vom System ausschließlich sequenziell gelesen werden [BBD+02]. Anfragen, welche solche Datenquellen verarbeiten, werden i. Allg. *kontinuierliche Anfragen* genannt und können in DBMS bspw. durch Trigger realisiert werden und stehen in spezialisierten Datenstrommanagementsystemen (DSMS) unmittelbar zur Verfügung [BBD+04]. Eine kontinuierliche Anfrage kann – ähnlich zu Planoperatoren zu DBMS – als ein gerichteter, azyklischer Operatorgraph interpretiert werden. Die Knoten beschreiben Operatoren, welche einzelne Verarbeitungsschritte repräsentieren. Die Kanten stehen für die Datenströme zwischen den Operatoren.

Für kontinuierliche Anfragen reichen die etablierten Konzepte zur Lastkontrolle jedoch nicht mehr aus, da permanent laufende Anfragen nicht nur zu Beginn der Verarbeitung überprüft werden müssen. Änderungen im Datenlieferungsverhalten von Quellen können zu kritischen Lastsituationen führen – auch ohne dass der Nutzer neue Anfragen hinzufügt. Zudem sollte für jedes Verarbeitungssystem die Lastkontrolle aufgrund unterschiedlicher Anwendungskontexte und Hardware individuell konfiguriert werden können.

In dieser Arbeit wird ein Konzept für eine flexible und erweiterbare AC-Komponente für kontinuierliche Anfragen vorgestellt, die sogenannte *kontinuierliche AC*. Grundlage ist ein dynamisches Regelwerk in Form des Event-Condition-Action (ECA)-Modells. *Events* (Ereignisse) sind in diesem Fall Änderungen im Systemzustand. *Actions* (Aktionen) sind die Maßnahmen, die die AC zur Lastkontrolle und -steuerung ausführen kann. Mit dieser Vorgehensweise ist es möglich, eine AC-Komponente zu entwickeln, die den besonderen Rahmenbedingungen der kontinuierlichen Anfragen gewachsen ist. Ziel dieser Arbeit ist es somit, die Möglichkeit eines ECA-Modells zur Umsetzung einer flexiblen, kontinuierlichen Admission Control aufzuzeigen. Die entwickelte Sprache CADL (Countinuous Admission Control Language) erlaubt die Definition komplexer Ereignisse und Regeln, um die Lastkontrolle individuell zu konfigurieren. Es ist nicht Ziel dieser Arbeit, eine konkrete AC für eine konkrete Anwendungsdomäne vorzustellen.

Die vorliegende Arbeit ist wie folgt gegliedert: Im Folgenden Abschnitt 2 wird das Konzept der kontinuierlichen AC genauer erklärt. In Abschnitt 3 wird die Evaluation des Konzepts erläutert. Anschließend werden in Abschnitt 4 die verwandten Arbeiten aufgelistet und beschrieben. Den Abschluss bildet Abschnitt 5 mit einer Zusammenfassung dieser Arbeit.

2 Konzept einer kontinuierlichen Admission Control

Kontinuierliche Anfragen können im Laufe ihrer Ausführung unterschiedliche Zustände einnehmen, welche sich je nach Verarbeitungssystem unterscheiden. Je nach Zustand erzeugt eine kontinuierliche Anfragen unterschiedliche Mengen an Systemlast. Abbildung 1 zeigt ein Beispiel für übliche Zustände und deren Übergänge. Im Allgemeinen ist jede neue kontinuierliche Anfrage zunächst *inaktiv*, d. h., sie ist installiert, verarbeitet jedoch keine Daten und erzeugt keine Systemlast. Eine solche Anfrage kann durch *starten* in den Zustand *aktiv* übergehen: Daten werden empfangen, verarbeitet und die Ergebnisse

ausgegeben. Dadurch steigt auch die Auslastung des Systems. Durch *stoppen* kann die Anfrage wieder auf *inaktiv* gesetzt werden. In den meisten Systemen können Anfragen ohne Berücksichtigung des Zustands jederzeit komplett entfernt werden.

Einige Systeme erlauben es, Anfragen zu pausieren, d. h., die Daten werden weiterhin von den Quellen empfangen, zwischengespeichert, aber nicht verarbeitet. Dadurch sinkt die Systemlast, jedoch wir mit zunehmender Dauer mehr Arbeitsspeicher zur Zwischenspeicherung benötigt. Alternativ kann die Last durch das Verwerfen von Daten reduziert werden (*partielle* Ausführung, sog. *Load Shedding* [TcZ+03]). Gegebenenfalls sind hier Kombinationen denkbar (z. B. *partiell pausiert*), d. h., es werden nicht alle Daten zwischengespeichert.

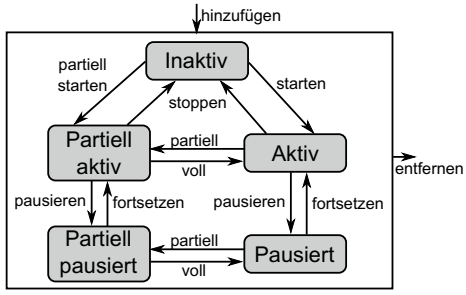


Abbildung 1: Mögliche Zustände für eine kontinuierliche Anfrage.

Die Systemlast für solche kontinuierliche Anfragen setzt sich i. d. R. aus den Systemressourcen Prozessor, Arbeitsspeicher sowie Netzwerkdatenrate zusammen. Die ersten beiden ergeben sich direkt aus der Tatsache, dass die Daten lediglich im Arbeitsspeicher zwischengespeichert und sofort verarbeitet werden. Aus diesem Grund wird hier externer Speicher vernachlässigt. Die Netzwerkbandbreite ist bei im Netzwerk verteilten Anfragen entscheident. Wird von einer Systemressource eine größere Menge benötigt, als von der Maschine zur Verfügung gestellt wird, tritt eine Überlastsituation ein. Eine bevorstehende Überlastung kann beispielsweise dadurch ankündigt werden, dass eine kontinuierliche Anfrage ihren Zustand wechselt (bspw. gestartet wird) oder sich die Datenrate einer Datenquelle ändert. Weiterhin können sich die Datenwerte ändern, sodass sich die Selektivität einiger Operatoren verändert und mehr Elemente als zuvor verarbeiten werden müssen. Werden umgekehrt Systemressourcen wieder frei, so können bspw. zuvor gestoppte kontinuierliche Anfragen wieder gestartet werden oder aproximierende Algorithmen durch exaktere erstetzt werden.

Die Idee ist, Indikatoren der Überlastung und geringere Systemlasten als Ereignisse aufzufassen. Diese geben bekannt, wenn eine kontinuierliche Anfrage den Zustand ändert, wenn größere Fluktuationen in den Datenraten auftreten oder wenn bestimmte Systemauslastungen über- oder unterschritten werden. Immer wiederkehrende Ereignisse können genutzt werden, um kontinuierliche Anfragen regelmäßig neu zu überprüfen.

Grundlage des hier vorgestellten Konzeptes für die kontinuierliche AC ist es, diese Ereignisse zur Laufzeit zu erfassen, sie in Abhängigkeit zum aktuellen Systemzustand auszuwerten, (potenzielle) Chancen und Risiken zu erkennen und sie ggfs. mittels gezielter (individueller) Maßnahmen auszunutzen bzw. aufzulösen. Um solche Ereignisse effektiv in der kontinuierlichen AC verarbeiten zu können und gleichzeitig Flexibilität zu gewährleisten, wird in dieser Arbeit ein ECA-Modell eingesetzt. Das Modell erlaubt die Definitionen der Reaktionen und Maßnahmen anhand von *Regeln*, welche auftretende Ereignisse auswerten. Sie werden zu sogenannten *Regelsätzen* zusammengefasst.

2.1 CADL als Sprache zur Regeldefinition

Zur einfachen Definition der Regeln wird die Beschreibungssprache CADL vorgeschlagen, die speziell auf die Lastkontrolle ausgelegt ist. Dort werden Ereignisse, welche vom System gesendet werden, als *atomare Ereignisse* bezeichnet. Jedes Ereignis hat einen Typ mit unterschiedlichen Attributen. Beispielsweise könnte ein Ereignis `QueryActivatedEvent` im Attribut `query` die Anfrage beinhalten, die gerade aktiviert wurde (und das Ereignis ausgelöst hat). CADL nutzt die Punktnotation, um auf Attribute eines Ereignisses zuzugreifen. Eine Regel lässt sich in CADL wie folgt definieren:

```
ON EVENT <ETyp> <EVariable> IF <Bedingung> THEN <Aktionen>
```

Tritt das Ereignis ein und die Bedingung ist erfüllt, werden die Aktionen ausgeführt. In den Bedingungen kann mittels der Ereignisvariable EVariable auf Attribute des konkreten Ereignisses zugegriffen werden. Bedingungen sind i. d. R. herkömmliche logische Ausdrücke, die wiederum aus mehreren Ausdrücken bestehen können, die mit `OR` oder `AND` verknüpft werden. Logische Ausdrücke beinhalten die typischen Bestandteile wie Funktionen, Konstanten, mathematische Operatoren und Vergleichsoperationen. Weitere Ereignisse können basierend auf anderen Ereignissen ausgelöst werden. Diese *komplexen Ereignisse* werden in CADL wie folgt definiert:

```
DEFINE COMPLEX EVENT <Ereignisname>:  <Kontinuierliche Anfrage>.
```

Damit kann `<Ereignisname>` in den Regeln ausgewertet werden. Die Idee ist, kontinuierliche Anfragen zur Identifikation der komplexen Ereignisse einzusetzen. Diese können die Ereignisse als Datenstrom empfangen und auswerten. Das Resultat ist eine Menge an komplexen Ereignissen, welche unter `<Ereignisname>` ansprechbar sind. Der Vorteil ist, dass der Nutzer auf der einen Seite den vollen Sprachumfang des Verarbeitungssystems nutzen kann, um komplexe Ereignisse zu definieren, auf der anderen Seite die kontinuierliche AC bereits etablierte Komponenten des Systems wiederverwenden kann, wie bspw. den Optimierer. Dabei ist jedoch eine strikte Trennung von Anfragen von CADL und normalen Anfragen notwendig, um Manipulation der Anfragen der AC zu vermeiden.

Mit CADL ist es möglich, einfach und flexibel komplexe Ereignisse und Regelsätze zu definieren. Der Anwender kann eigene atomare Ereignisse, Bedingungen und Aktionen implementieren und in CADL einsetzen. Damit ist es möglich, das ECA-Modell für besondere Anwendungen zu spezialisieren. Es unterliegt jedoch der Verantwortung des Nutzers, für die Situation sinnvolle komplexe Ereignisse und Regeln zu definieren. Wie bei jedem ECA-System muss der Regelersteller auf konsistente Regeln achten. Dies soll hier jedoch nicht weiter betrachtet werden.

2.2 Architektur der kontinuierlichen AC

Der Kern des Konzeptes der kontinuierlichen AC besteht darin, einen mit CADL definierten Regelsatz zu nutzen, um auf atomare und komplexe Ereignisse zu reagieren und unter den gegebenen Bedingungen Aktionen zur Lastkontrolle durchzuführen. Die Architektur des Konzepts ist in Abbildung 2 zu sehen.

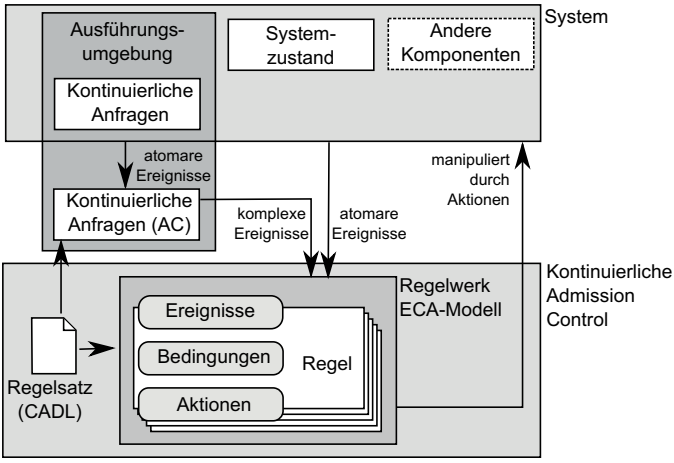


Abbildung 2: Konzeptionelle Übersicht der kontinuierlichen AC.

In der Ausführungsumgebung werden die (kontinuierlichen) Anfragen organisiert. Die Umgebung steuert und koordiniert die Ausführung der kontinuierlichen Anfragen und verwaltet ihre Zustände. Sie sendet i. d. R. atomare Ereignisse, wenn kontinuierliche Anfragen hinzugefügt werden, ihren Zustand ändern oder vom System entfernt werden. Im Regelsatz definierte kontinuierliche Anfragen werden gesondert behandelt: sie liefern keine atomaren Ereignisse, sondern verarbeiten diese zu komplexen Ereignissen, die bspw. bestimmte Zustandsverläufe erkennen und melden können. Zusätzlich senden kontinuierliche Anfragen selbst atomare Ereignisse, um Veränderungen in der Datenverarbeitung mitteilen zu können: Fluktuationen in der Datenrate können zu änderungen in der Systemlast führen, sodass solche atomare Ereignisse wichtig werden.

Es ist notwendig, dass das System Möglichkeiten anbietet, den eigenen Systemzustand (und insbesondere die Systemlast) zu beobachten. Atomare Ereignisse können hier bspw. überschreitungen von Lastschwellwerten sein. Es ist dann für die kontinuierliche AC möglich, auf Schwellwerte zu reagieren, auch wenn keine neue kontinuierliche Anfrage gestartet wurde. Schließlich können andere Komponenten des Systems atomare Ereignisse generieren (bspw. der Nutzer oder basierend auf Netzwerksignalen).

Alle Ereignisse werden im ECA-Modell des kontinuierlichen AC ausgewertet. Dazu werden im Regelsatz u. a. Bedingungen spezifiziert. Diese können neben typischen Ausdrücken, Konstanten und Funktionen auch spezielle Ausdrücke bzgl. der Lastüberwachung beinhalten (bspw. Abfrage der aktuellen Systemlast).

Das Resultat der Auswertung ist eine Menge von Aktionen. Diese Aktionen werden von der AC ausgeführt, um das System (und damit die Systemlast) zu manipulieren. Ein typischer Ansatz zur Senkung der Systemlast besteht in der Zustandsänderung der Anfragen, indem die kontinuierliche AC Anfragen stoppt, pausiert, partiell ausführt oder ganz entfernt (analog zu den Zuständen in Abbildung 1). Im Netzwerk könnte die AC einzelne Anfragen an andere Maschinen abgeben. Es sollte jedoch berücksichtigt werden, die Ausführung so wenig wie möglich zu beschneiden. Deswegen sollten leichte Aktionen wie bspw. die partielle Ausführung bevorzugt werden. Nur in kritischen Situationen soll-

ten Anfragen gestoppt oder sogar ganz entfernt werden. Später können Regeln erkennen, ob Leistungsreserven wieder verfügbar sind. Je nach System können kontinuierliche Anfragen wieder gestartet oder – falls zuvor partiell ausgeführt – wieder alle Daten verarbeiten. Eine gestaffelte Vorgehensweise zur Reaktivierung der kontinuierlichen Anfragen kann hier ebenfalls nützlich sein. Anstatt eine gestoppte Anfrage wieder vollständig auszuführen, kann sie zunächst partiell ausgeführt werden. Dabei können weitere Rahmenbedingungen wie bspw. die Genauigkeit der Ergebnisse berücksichtigt werden.

3 Evaluation

Das vorgestellte Konzept wird anhand eines einfachen Szenarios evaluiert, mit dem gezeigt wird, wie mit beispielhaften Regeln eine Überlastung des Systems verhindern wird. Fokus dieser Arbeit ist es nicht, ein komplexes Regelwerk zu evaluieren, da diese von der Hardware, der Anwendungsdomäne sowie der eingesetzten Software abhängig ist.

Zur Umsetzung der kontinuierlichen AC wurde das Framework für DSMS Odysseus verwendet [AGG+12]. Seine komponentenbasierte Architektur erlaubt die Implementierung zusätzlicher Funktionen, ohne die Basiskomponenten zur Verarbeitung anpassen zu müssen. Die kontinuierliche AC mitsamt dem ECA-Modell und der Sprache zur Regeldefinition wurde dort als zusätzliche Komponente umgesetzt. Zur Laufzeit sendet Odysseus Ereignisse bzgl. der Ausführung der kontinuierlichen Anfragen und der Systemlast.

Die genutzte Maschine ist ein Notebook mit Windows 7 als Betriebssystem, hat 4 GB RAM und einen Intel Core 2 Duo CPU T9500 Prozessor mit 2,60 GHz. Neben Odysseus laufen auf der Maschine keine weiteren Applikationen. Es wurde bewusst eine relativ leistungsschwache Maschine ausgewählt, um die Effekte der Lastkontrolle besser hervorheben zu können.

Während der Ausführung von Odysseus wird alle zehn Sekunden eine neue kontinuierliche Anfrage gestellt und ausgeführt (ohne die alten kontinuierlichen Anfragen zu stoppen). Dies geschieht bis zu einer Menge von 20 kontinuierlichen Anfragen. Damit wird die Lastgrenze der genannten Maschine erreicht. Für die Reproduzierbarkeit der Evaluation ist jede Anfrage gleich, Optimierungen sind deaktiviert. Die Anfrage besteht hier aus speziellen Benchmarkoperatoren und erfüllt lediglich den Zweck, reproduzierbar eine bestimmte, konstante Prozessorlast zu erzeugen. Durch das permanente Hinzufügen neuer Anfragen erhöht sich mit der Zeit die gesamte Systemlast.

Zur Laufzeit wird die Prozessorauslastung alle drei Sekunden mittels systeminterner Methoden gemessen. Ein Wert von 100% entspricht dabei einer vollen Auslastung. Da ausreichend Arbeitsspeicher vorhanden ist und um die Evaluationsergebnisse einfach zu halten, wird auf eine Messung der Belegung des Arbeitsspeichers verzichtet. Sie wäre jedoch im Rahmen des vorgestellten Konzeptes problemlos möglich. Weiterhin werden im Laufe der Experimente Latenz und Datenrate gemessen. *Latenz* beschreibt die Zeit, die benötigt wird, um mit Hilfe eines eingehenden Datenelements ein Ergebnis zu produzieren. Das bedeutet für dieses Szenario, wie lange ein Element im System verbleibt. *Datenrate* beschreibt die Menge an verarbeiteten Datenelementen pro Sekunde.

3.1 Ergebnisse

Das erste Experiment soll zeigen, dass durch das ständige Hinzufügen neuer kontinuierlicher Anfragen die Maschine (und damit Odysseus) überlastet werden kann, wenn die kontinuierliche AC nicht eingreift. Das Resultat der regelmäßigen Messung der Prozessorauslastung ist in Abbildung 3 als gestrichelte Linie zu sehen.

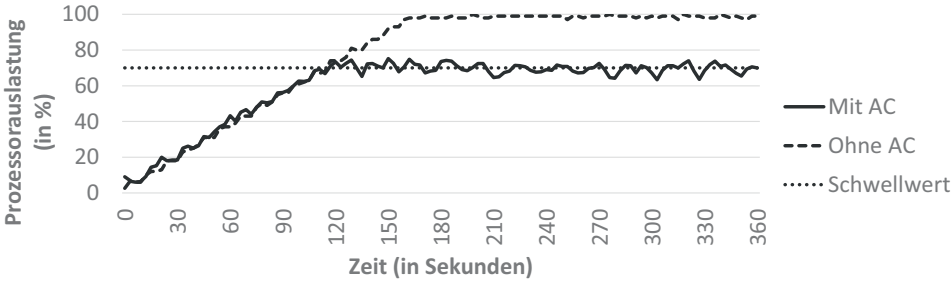


Abbildung 3: Prozessorauslastungen ohne Regeln (gestrichelte Linie) und mit (durchgezogene Linie)

Die Last steigt mit zunehmender Menge an aktiven kontinuierlichen Anfragen, bis schließlich nach ungefähr 150 Sekunden das Maximum an Prozessorlast erreicht wurde und Odysseus als überlastet gilt.

In einem zweiten Experiment soll die kontinuierliche AC eingesetzt werden, um die Überlastsituation zu vermeiden. Dazu wurde mittels CADL ein Regelsatz bestehend aus mehreren Regeln und einem komplexen Ereignis definiert. Der Regelsatz stoppt neue kontinuierliche Anfragen, wenn die aktuelle Prozessorlast 70% übersteigt. 30% der Leistung soll als Reserve vorgehalten werden (bspw. für Lastspitzen). Zusätzlich wird zwischen wichtigen kontinuierlichen Anfragen (Priorität $>= 10$) und unwichtigen (Priorität < 10) unterschieden. Erst sollen niedrig priorisierte Anfragen gestoppt werden, bevor es hoch priorisierte Anfragen betrifft. Können Anfragen später wieder reaktiviert werden, sollen hoch priorisierte Anfragen bevorzugt werden. Die Regeldefinition beginnt wie folgt:

```
ON EVENT QueryActivatedEvent e
IF curCpuLoad() > 70 AND e.query.prio < 10
THEN stopQuery( e.query )
```

Damit werden niedrig priorisierte Anfragen sofort gestoppt, wenn die Prozessorlast aktuell 70% übersteigt. Bei neuen hoch priorisierten Anfragen muss mittels zwei weiteren Regeln anders vorgegangen werden:

```
ON EVENT QueryActivatedEvent e
IF curCpuLoad()>70 AND e.query.prio >= 10
   AND existsQuery(state=ACTIVE AND prio<10)
THEN stopQuery(selectRandomQuery(state=ACTIVE AND prio<10))

ON EVENT QueryActivatedEvent e
IF curCpuLoad()>70 AND e.query.prio>=10
   AND NOT existsQuery(state = ACTIVE AND prio < 10)
THEN stopQuery(e.query)
```

Bei neuen hoch priorisierten Anfragen wird anstelle der neuen Anfrage eine bereits aktive niedrig priorisierte Anfrage gestoppt (erste Regel). Ist eine solche Anfrage nicht vorhanden (oder alle sind bereits gestoppt), muss die hoch priorisierte Anfrage gestoppt werden

(zweite Regel). Die Funktion `existsQuery` überprüft, ob eine Anfrage im System existiert, die das genannte Prädikat erfüllt (hier: ob eine Anfrage aktiv ist und eine Priorität kleiner als zehn besitzt). Die Funktion `selectRandomQuery` funktioniert ähnlich, außer dass aus der Menge der kontinuierlichen Anfragen, welche das Prädikat erfüllen, genau eine zufällig ausgewählt und zurückgegeben wird.

Werden Anfragen gestoppt, sinkt die Systemlast, sodass es auch die Chance gibt, zuvor gestoppte Anfragen wieder zu starten. Sei dazu `TryRestartQueryEvent` ein Event, welches alle 10 Sekunden aktiviert wird:

```
ON EVENT TryRestartQueryEvent e
IF curCpuLoad()<70 AND existsQuery(state=INACTIVE AND prio>=10)
THEN startQuery(selectRandomQuery(state=INACTIVE AND prio>=10))

ON EVENT TryRestartQueryEvent e
IF curCpuLoad()<70 AND existsQuery(state=INACTIVE AND prio<10)
   AND NOT existsQuery(state=INACTIVE AND prio>=10)
THEN startQuery(selectRandomQuery(state=INACTIVE AND prio<10))
```

Das komplexe Ereignis `TryRestartQueryEvent` wird alle zehn Sekunden automatisch ausgelöst, sodass die beiden Regeln regelmäßig unabhängig vom Zustand des Systems geprüft werden. Die erste Regel besagt, dass bei einer niedrigen Prozessorlast (unter 70%) eine hoch priorisierte Anfrage gestartet werden kann, falls eine solche Anfrage gestoppt im System vorliegt. Ist dies nicht der Fall und es existieren gestoppte niedrig priorisierte Anfragen, wird eine davon gestartet (zweite Regel). Bei der Durchführung des Experiments sind die ersten 14 Anfragen niedrig priorisiert. Anschließend werden sechs hoch priorisierte Anfragen hinzugefügt. Das Resultat der Messungen der Prozessorlast im zweiten Experiment ist in Abbildung 3 als durchgezogene Linie dargestellt.

Die gepunktete Linie beschreibt die 70%-Marke. Man erkennt, dass analog zum Vorversuch die Prozessorlast mit zunehmender Menge an kontinuierlichen Anfragen steigt, sich jedoch um den Wert 70% einpendelt. Dies hat den Grund, dass ab 70% durch die ersten Regeln neue Anfragen nicht mehr aktiv werden. Kommen später die hoch priorisierten Anfragen hinzu, werden andere niedrig priorisierte Anfragen gestoppt, um die Prozessorlast um 70% zu halten. Fällt die Last unter 70%, werden gestoppte Anfragen wieder ausgeführt, wobei hoch priorisierte Anfragen bevorzugt werden. Durch die zufällige Auswahl einer Anfrage wird über die Zeit jede Anfrage irgendwann ausgeführt.

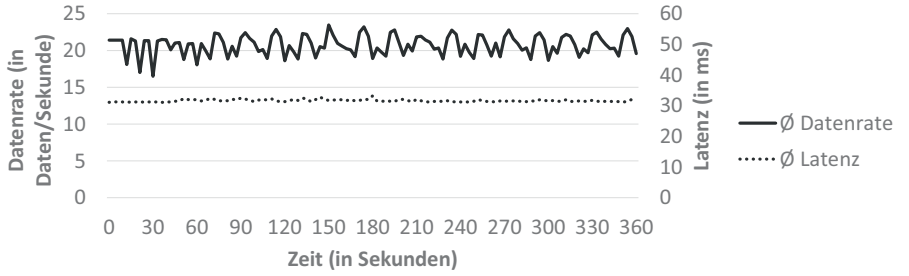


Abbildung 4: Verlauf der durchschnittlichen Datenrate und Latenz mit Regeldefinition.

Abbildung 4 zeigt die gemessenen Datenraten und Latenzen im zweiten Experiment. Die durchschnittliche Datenrate der Anfragen (durchgezogene Linie) bleibt auf gleichem Ni-

veau. Weiterhin ist die Latenz stabil. Das bedeutet, dass das System die Verarbeitung weiter effizient und effektiv fortsetzen kann. In realen Szenarien sollten jedoch andere Aktionen als das Stoppen gewählt werden (bspw. die Auslagerung im Netzwerk).

Die beiden Experimente zeigen, dass das hier vorgestellte Konzept seine Aufgabe der Lastkontrolle erfüllen kann: während noch im ersten Versuch ohne Einsatz des kontinuierlichen AC schnell eine Überlastsituation entstand, konnte diese im zweiten Versuch mittels einer relativ einfachen Regeldefinition abgewendet werden.

4 Verwandte Arbeiten

Wie bereits zu Beginn erwähnt, wird die AC in vielen konventionellen DBMS dann eingesetzt, wenn eine neue Anfrage ausgeführt werden soll [HSH07]. Diese Vorgehensweise der AC ist in vielen DBMS ausreichend, erlaubt jedoch nicht eine einfache und flexible Regeldefinition wie in dieser Arbeit vorgeschlagen.

Für DSMS werden in der Literatur hauptsächlich konkrete Ansätze zur Lastreduktion vorgestellt und diskutiert. Ziel ist es oft, eine maximale Genauigkeit mit minimalem Ressourcenverbrauch zu erreichen, wie es bspw. ausführlich in [ABB+04] beschrieben wird. Im Fokus der Literatur steht hauptsächlich das konkrete Problem des begrenzten Arbeitsspeichers. In [BBD+02] werden diverse Techniken beschrieben. Mittels des Fensteransatzes wird nur ein Teil des Datenstroms als gültig markiert (und verarbeitet). Mittels Sampling und Synopsen werden Datenströme verdichtet. Letztere werden bspw. in STREAM eingesetzt [MWA+03]. Load Shedding [TcZ+03] wird in vielen DSMS erfolgreich eingesetzt und wird bereits in dieser Arbeit als ein möglicher Zustand einer kontinuierlichen Anfrage betrachtet. Kalman-Filter werden genutzt, um zukünftige Systemlasten vorherzusagen um damit präventiv Maßnahmen ergreifen zu können [JCW04]. Mittels einer intelligenten Operatorreihenfolge lassen sich Systemlasten ebenfalls reduzieren [BBD+04].

5 Zusammenfassung

In Datenbanksystemen ist es wichtig, dass eine reibungslose Verarbeitung der Daten jederzeit sichergestellt ist. Überlastungen des Systems können zu Störungen und Ausfällen führen. Für DBMS existieren bereits Komponenten zur Lastkontrolle, die sogenannte Admission Control (AC). Jedoch existieren Anwendungsszenarien, die eine kontinuierliche und zeitnahe Verarbeitung erfordern. Das bedeutet, dass eine kontinuierliche Anfrage permanent ausgeführt werden muss, um sofort auf eintreffende Datenelemente einer aktiven Datenquelle reagieren zu können.

Für kontinuierliche Anfragen ist eine konventionelle AC nicht ausreichend: Sie muss nicht nur Anfragen zu Beginn, sondern auch während ihrer Ausführung prüfen, da durch die kontinuierliche Verarbeitung sich die Systemlast ständig ändern kann. Durch die Vari-

anz an Datenquellen, verschiedenen DBMS, DSMS und unterschiedlichen Hardwarekapazitäten ist eine Beschreibung für ein allgemeingültiges Verhalten nicht sinnvoll.

In dieser Arbeit wurde eine *kontinuierliche AC* auf Basis eines ECA-Modells vorgestellt, welche in Systemen mit kontinuierlichen Anfragen eingesetzt werden kann, um die dortige Systemlast permanent zu kontrollieren und ggfs. zu regulieren. Vom System gesendete, atomare Ereignisse können mittels Regeln ausgewertet werden und so zu gezielten Reaktion zur Lastkontrolle führen. Durch die vorgestellte Sprache CADL lassen sich flexibel komplexe Ereignisse und Regelsätze definieren, die an die speziellen Anforderungen und Rahmenbedingungen für kontinuierliche Anfragen angepasst sind. Die Evaluation zeigt, dass das Konzept praktikabel, umsetzbar und funktionsfähig ist.

Literatur

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister und Jennifer Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM Trans. Database Syst.*, 29(1):162–194, Marz 2004.

[AGG+12] H.-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen und Daniela Nicklas. Odysseus: a highly customizable framework for creating efficient event stream management systems. DEBS '12, Seiten 367–368, New York, NY, USA, 2012. ACM.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Jennifer Widom. Models and issues in data stream systems. PODS '02, Seiten 1–16, New York, NY, USA, 2002. ACM.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Dilys Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, Dezember 2004.

[CKSV08] M. Cammert, J. Kramer, B. Seeger und S. Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *Knowledge and Data Engineering, IEEE Transactions on*, 20(2):230–245, Feb 2008.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker und James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[JCW04] Ankur Jain, Edward Y. Chang und Yuan-Fang Wang. Adaptive Stream Resource Management Using Kalman Filters. SIGMOD '04, Seiten 11–22, New York, NY, USA, 2004. ACM.

[Krä09] Jürgen Krämer. Continuous Queries over Data Streams - Semantics and Implementation. Jgg. 144 of *BTW '09*, Seiten 438–448. GI, 2009.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein und Rohit Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[TcZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack und Michael Stonebraker. Load Shedding in a Data Stream Manager. VLDB '03, Seiten 309–320. VLDB Endowment, 2003.

Schema Engineering

Meeting the Challenges of Integrating Large and Diverse Geographic Databases

Michael Schäfers, Udo W. Lipeck

Institut für Praktische Informatik, Fachgebiet Datenbanken und Informationssysteme
Gottfried Wilhelm Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover
{schaefers,lipeck}@dbs.uni-hannover.de

Abstract: Using data matching techniques to identify multiple representations of the same real-world entity is an essential step for all data integration tasks. While matching standard data types like strings or numbers with generic methods is well-studied, approaches for non-standard data have to deal with domain-specific challenges. For geographic databases containing spatial features we face a high degree of diversity in terms of geometric and semantic modeling between data sources. Likewise, complex geometric data types and topological relations require efficient processing. Finally, geodatabases can grow very large if they cover extensive regions or whole countries.

In this paper, we present our SimMatching approach for integrating relational geodatabases that meets these challenges. In particular, we study road networks from several data sources. Our iterative algorithm matches semantically equivalent objects based on geometric and semantic attribute similarity measures. Relational similarity helps to solve difficult situations by exploiting the underlying graph structure of road networks: Already confirmed neighbouring matchings improve the similarity value of a given matching. Adaptability to diverse input data is reached by combining and weighting subsets of similarity measures. A greedy approach and an efficient end-to-end-system built upon simple and flexible components outperform previous systems in terms of runtimes while showing matching results of high quality. Scalability to large geodatabases is supported by a partitioning framework together with parallel processing. We have experimentally verified our approach with large real-world datasets.

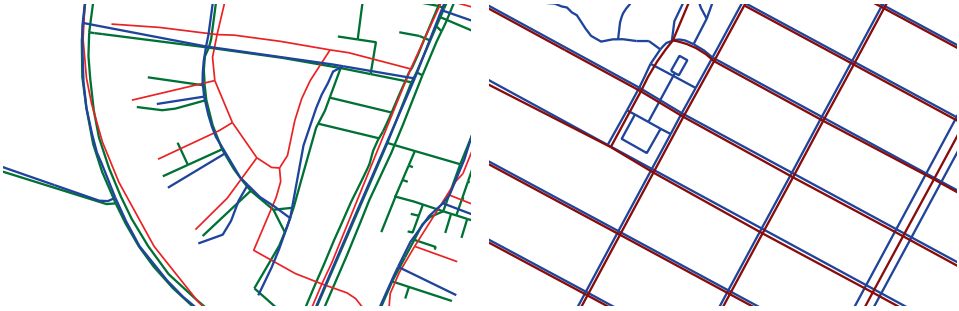
1 Spatial Data Integration and its Challenges

Spatial data are nowadays collected and maintained by numerous organizations. Despite the complexity of establishing an own spatial dataset, this task is motivated by a variety of reasons: Public authorities have to fulfill their public contract like establishing a cadastral information system with exact spatial reference data. Companies see the economic advantages of having a proprietary dataset that specializes to their own application requirements and outperforms competitors. Finally, an open data project like *OpenStreetMap (OSM)* aims at making geographic information freely available for everyone.

These different stakeholders and their motivations result in a high degree of diversity between spatial databases. While federal data aims at high spatial exactness and complete-

ness, datasets for creating printed maps intentionally abandon these aspects in favor of better readability. Here, *cartographic generalization* is used to simplify the real world and to align objects for an eye-friendly visualization, e.g., on paper or small screens. Finally, car navigation systems are highly depending on reliable semantic attributes, e.g., speed limits or turning restrictions, for a safe and efficient routing. All in all, differences in spatial exactness, completeness, currentness, availability and quality of attributes, etc. should be considered when analyzing and comparing geodatabases.

For example, in figure 1, geometric differences between multiple sources can be observed in visualizations of small map excerpts. A complex road network together with severe differences in object modeling shown in figure 1a makes an object-wise comparison obviously difficult, even though the same real-world situation is pictured. However, for figure 1b (showing a simple and uniform road network) this task is significantly easier. We conclude that comparability between input data highly depends on the source characteristics.



(a) Complex and diverse (Munich; OSM, ATKIS, D20) (b) Simple and uniform (Manhattan; OSM, TIGER)

Figure 1: Comparison of road networks from different geographic data sources.

With this **high degree of diversity**, we have identified one of the main challenges for *integrating* multiple geodatabases. However, this aspect is also the main cause for benefits of successful spatial data integration. The more differences we can spot between input sources, the more possibilities arise to enhance the sources with each other's information: After identifying and linking different representations of the same real-world entity, *information fusion* can be used to enhance one dataset with information from another one. E.g., speed limits don't have to be collected by a time-consuming exploration of the actual roads, but can be easily transferred from an existing database already containing this information. Next, *change detection* can be used to update contents when sources of different currentness are integrated. Objects that cannot be matched reveal added, removed or updated objects in either dataset.

As seen in figure 1a, different representations of the same real-world entity usually cannot be identified obviously. Spatial data lack global identifiers like ISBN numbers for books, and there is no notion of equality between database objects. Thus, we have to identify *semantically equivalent* objects based on *similarity* between objects in order to create an instance-level matching. A combination of several *similarity measures* has to be applied to classify objects as matching or non-matching based on their degree of similarity.

An algorithm for matching spatial data of arbitrary sources has to be **adaptable** to input sources by allowing flexible combinations of similarity measures which correspond to the source characteristics. Since every spatial feature has a geometry attribute, various measures apply geometric computations. Working with geometric attributes is far more **complex** than handling strings or numeric data types. In order to achieve low runtimes, spatial matching algorithms have to operate very **efficiently**. Keeping the overall algorithmic structure, the measures, and other components **simple**, also helps to improve efficiency and to guarantee fast processing of datasets. Finally, spatial databases are usually **very large**. They can easily grow to millions of objects when spatial features like roads, railways, or buildings for a whole country are considered. Therefore, approaches for matching large datasets need to be extremely **scalable** to handle this amount of data.

In this paper, we present our SimMatching algorithm to match spatial databases containing road networks. We meet the previously identified challenges of spatial data integration with the three main goals of our algorithm:

1. **Adaptability** to **diverse input data**: Our algorithm highly relies on similarity between database objects. Appropriate similarity measures can be chosen and combined with respect to the input source characteristics. Next to semantic and geometric attribute-based measures, we also introduce a relational measure to incorporate context information to the matching process. Similarity of a possible matching rises in our iterative approach, the more directly connected objects have already been matched in previous iterations.

2. **Simplicity** and **efficiency** to process **complex objects**: We propose a greedy approach [CLRS09] for the best trade-off between short runtime and high result quality. Our approach is built from simple components tailored together in a straightforward iterative process with a clear algorithmic structure. We leave geometric computations to tailored libraries available for in-memory programming, thus avoiding non-optimized DBMS extensions. Of course, large datasets are provided by database systems. Results (i.e., links between objects) are stored there, too.

3. **Scalability** to **large datasets**: A partitioning framework supports our in-memory algorithm by splitting input data in manageable pieces. Thus, processing large datasets is feasible even with limited computation resources (e.g., main memory). Processing disjoint partitions in parallel, as well as optimizations to keep the number of match candidates low help to achieve scalability.

While existing approaches usually address only one or two of these goals, our contribution is an end-to-end system that is optimized for all three concerns. In the remainder of this paper, we first survey related work and position our approach in the next section. Section 3 introduces our algorithm with its strong key concepts, i.e., flexible similarity measures, well-dosed consideration of relational aspects, efficient blocking techniques, and partition-wise parallel processing. In section 4 we evaluate runtimes and result quality with real-world datasets. We show that our algorithm outperforms previous approaches in terms of runtimes and efficiency while delivering results of equally high or even better quality. Finally, we conclude this paper in section 5 and give a short outlook on future work.

2 Related Work

Information integration is a well-studied research area in the database community. Approaches to integrate *standard data* that contains strings or numeric values for, e.g., national census, the health sector (patient records), or business applications (customer records) reach back to the 1950s. Leser and Naumann [LN07] as well as Doan et al. [DHI12] provide comprehensive compendia on integrating distributed and heterogeneous data sources. The authors identify data integration problems (e.g., structural, semantic, and other conflicts) and present established solutions for creating a redundancy-free integrated dataset (e.g., schema and data matching). They also present relevant system architectures like *federated databases* which are introduced in detail by Conrad [Con97]. Several concepts of federated databases are considered for the system architecture presented in our paper. E.g., we keep the input sources autonomous and untouched by the integration process.

Rahm and Bernstein survey techniques for automatic *schema matching* [RB01]. Though, as our geographic data sources are structured rather simply with few attributes[1] we rely on manual schema matching. Thus, the task of *data matching* is more relevant for our work. Christen [Chr12] provides a great book on this research area. The whole matching process with its different steps like preprocessing, indexing/blocking, data comparison, and classification is explained. However, most techniques concentrate on standard data.

Most matching techniques only consider attributes of an object collection. However, several approaches prove that *relations* between objects can be very valuable for matching tasks. The clustering algorithm for relational entity resolution by Bhattacharya and Getoor [BG07] exploits relations between cooccurring entities to support attribute-based matching. E.g., for bibliographic data, relations exist between papers and authors. Two representations of the same author are more likely to form a valid matching if the connected papers have already been successfully matched before. Next, Melnik et al. [MGR02] exploit relations from a graph representation for schema matching. Their similarity flooding uses a fixpoint computation to propagate similarity values to adjacent graph nodes. I.e., results gathered in earlier iterations influence connected objects. We propose a relational similarity measure for spatial data that is inspired by both papers.

Approaches for automatic spatial data integration have also been proposed in the context of *geographic information systems (GIS)* since the 1980s. Saalfeld [Saa88] coin the term *map conflation* for matching and aligning two geographic maps. Conflation results in a single map visualization that contains features and fused attributed of both maps. The matching task is focused on point features (road intersections) to find reference points for a local rubber-sheeting transformation. The latter aligns map features to the reference dataset by moving or distorting their geometries. The same authors extend their approach to consider semantic (if available) and topology information to improve the matching of corresponding point features. Line features (e.g., roads or rivers) are more significant for most maps than points. Therefore, several approaches propose to use their geometries as anchors, e.g., as in [DFE01]. This conflation approach exploits various criteria (e.g., shape and distance) to match line features.

[1] Most likely because ESRI shapefiles, a de-facto standard for file-based storage and exchange of spatial data, only allow a simple relational structure with one single attribute table.

All these classic map conflation approaches seem optimized for two fixed input sources. They are also primarily based on "visual" observations (derived from a map representation). The feature aligning is usually highly tailored to the matching step. With the possibility to keep several datasets in an integrated database instead of having to align them for a single map visualization, we strive to follow the architecture of a *federated database* [Con97] and prefer to leave input sources unchanged. Matchings between database objects representing the same real-world entity are determined by links between their original features. This also allows transferring geometries along links in either direction, though in a separate later fusion step.

With the wider availability of structured attributed vector datasets, Walter and Fritsch [WF99] propose a statistical approach for matching road networks. A growing buffer around a given road object identifies possible match partners (other roads lying within this buffer area). Several geometric characteristics are then employed to find a unique set of final matchings. The authors also consider matchings of higher cardinality than *1:1*, i.e., *n:m*-matchings: Objects have to be aggregated for matching, if sources differ in topology. The matching quality is satisfying for this approach, but very long runtimes make processing full real-world sources in reasonable time unfeasible.

Relying only on attribute-based characteristics for matching objects often leads to critical situations. Multiple nearby objects (e.g., parallel roads with similar shape) may lead to ambiguous matchings if a matching algorithm only considers a local perspective with autonomous objects. Context information – usually gathered from the dataset topology – can be incorporated into the matching process to improve matching results. Zhang [Zha09] suggests to follow continuous polylines, so-called "strokes", in road networks. Here, fewer geometric comparisons and no semantic information are exploited, but the graph structure of the dataset is of high importance. However, algorithms and data structures are highly optimized for specific sources and seem rarely adaptable to diverse input data.

Several other approaches are based on polylines as well. E.g., Safra et al. [SKSD13] concentrate on high performance with their "ad hoc" matching. They achieve low runtimes by simplifying line geometries to straight line representations and matching line endpoints only. Though, from our experience, such simple geometric comparisons just work for simple and homogeneous road networks. Lots of valid matchings are missing if generalization or missing crossroads lead to different geometric object modeling between input data.

Mantel et al. [ML04] adapt and improve the original *buffer growing algorithm* by [WF99] to integrate federal and proprietary spatial data within a project funded by the German Federal Agency for Cartography and Geodesy. Their core idea is to catch possibly matching objects within a buffer around increasing aggregations of given objects. Similarity computation is based on attributes only: Aspects of the geometry (e.g., location or length) and semantic information (e.g., object classes) are considered. Because structural information is neglected completely, result quality is critical for nearby, similar-shaped lines.

Tiedge et al. propose a *graph matching algorithm* [TL07] that operates on the underlying graph structure carried by every road network. Thus, they consider relevant context information. First, they compose a product graph from both input graphs. Next, the product graph is reduced to an unambiguous set of resulting matchings by applying structural,

topological, geometric, or semantic rules. Thus, the algorithm considers attribute and relational (graph-based) similarities, but encodes them rather implicitly. Although this combination significantly enhances result quality, the dominance of graph handling leads to high runtimes and to a highly complex system of reduction rules.

Within the project, our group furthermore gathered some experience in handling larger datasets: A framework for partitioning and recomposition tasks is proposed by Warneke et al. [WL12]. Their techniques make processing data for a whole federal state of Germany feasible – though, with runtimes of several hours when using the graph matching algorithm.

Based on this know-how, we conclude that only a combination of geometric, semantic, *and* relational criteria can guarantee high quality matching results for integrating arbitrary datasets. Furthermore, efficient and simple components make processing large datasets feasible. We have also learned that spatial databases are slow for geometric computations and geometry-changing operations. Therefore, we rely on in-memory processing on client machines and use optimized geometric libraries. On the server side we use a spatially enhanced database management system to store input data and matching results. Spatial indexes support efficient spatial queries to access partitioned input data.

3 Similarity-based Spatial Matching

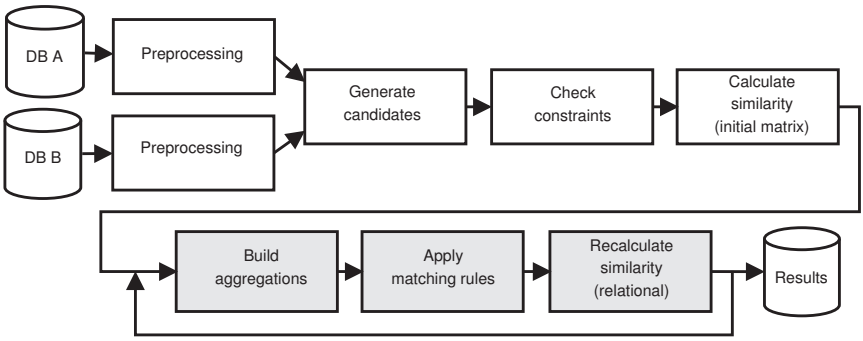


Figure 2: Flowchart of the overall SimMatching algorithm (without partitioning).

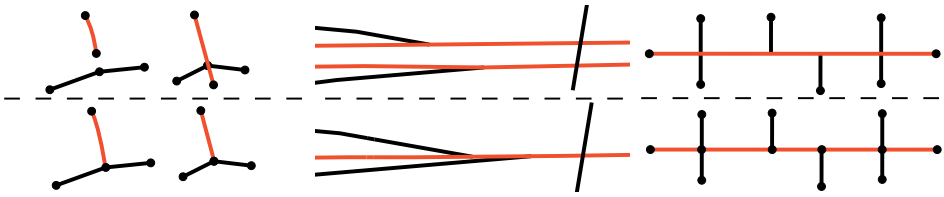
The *SimMatching* algorithm (figure 2) is mainly based on a selection of similarities between complex objects. To achieve both high efficiency and result quality, we have chosen a *greedy* strategy, i.e., choosing the locally optimal choice at each stage [CLRS09]. Our approach is also iterative to make use of previous results: Relational similarity rises, the more matchings in the vicinity of possible matchings are already confirmed as valid matchings. The algorithm is highly adaptable by configuring similarity measures, weights, thresholds, and other parameters. A partitioning framework is wrapped around the algorithm to split larger input data, process manageable pieces at a time, and recompose the results.

After preprocessing both input datasets, an initialization phase generates match candidates,

validates them by checking constraints, and calculates initial similarity values. Then the algorithm enters its iterative part: For the currently best possible matching we build aggregations of higher cardinality to see if similarity improves by doing so. Finally, we apply matching rules within a classification step to confirm or reject matchings with respect to their similarity values. Afterwards, subsequent recalculations of similarity might be needed. The algorithm ends, if no more matchings can be confirmed. Next, all components of the algorithm representing its single steps will be introduced in detail.

Preprocessing

Errors in the input data and different object modeling rules require a *preprocessing* before the actual matching algorithms starts. Our SimMatching approach assumes that the given road network represents a *connected planar graph* in order to take advantage of reliable topological information. We remove small gaps or overlappings and segment lines at intersections to meet this requirement. Special treatment of multi-lane-roads helps to further improve comparability. In detail, the following steps are done for both input sources separately:



(a) Topological correction. (b) Multi-lane replacement. (c) Line segmentation.

Figure 3: Original (top) and preprocessed (bottom) input data.

Topological errors like small gaps (undershoots) and overlapping line ends (overshoots) can occur, if data is captured or modeled inexactly (figure 3a). E.g., hand-drawn vector maps might be styled by visual aspects only. Errors remain invisible for human operators if they use coarse tools or thick visualizations to create the skeleton of road networks. However, such errors make a dataset unusable for, e.g., routing purposes, when connections between lines are missing. We correct these errors by harmonizing coordinates for line endpoints and merging nearby points.

Some larger roads, e.g., highways, are modeled with **multi-lane-representations** in some sources, while others just use **single-lane-representations** (figure 3b). Here, we unify modeling rules and the dataset structure by replacing occurrences of parallel lines (with "one-way" attributes in the opposite direction) by a middle axis representation. Meeting lines (i.e., crossroads and highway ramps or exits) are adjusted to the new topological structure after computing the middle axis.

Next, a **line segmentation** algorithm (figure 3c) splits longer lines into smaller segments. Intersecting lines have to meet in known graph nodes that are represented by line endpoints only. This line segmentation also helps to unify object modeling rules. Choosing equally defined "smallest parts" for all input data helps to improve comparability. Our segmen-

tation algorithm increases the number of preprocessed objects compared to the original dataset cardinality. If, e.g., two lines are intersecting each other, four segments are created as seen in figure 3c.

Finally, **neighbours** for every input object are calculated and stored for fast access during subsequent steps. We extract information about incident edges from the underlying graph representation of the road network. The resulting incidence lists are stored in an object-based representation in order to encode the graph structure.

Generate Candidates

The first step for every matching process is the generation of *match candidates*, i.e., pairs of objects which are considered possibly matching and therefore kept for further computations in subsequent steps. When matching two datasets of cardinality n and m respectively, the naive approach to build these candidates would be to calculate the *cartesian product* which means to compare every single object from the first source with every single object from the second source. Processing the complete resulting candidate set of size $n \times m$ is only feasible for small input data and does not scale to larger datasets. Thus, the amount of match candidates has to be reduced in the very beginning of the matching process.

Blocking techniques [Chr12] are applied to group the input objects into smaller subsets of the original data. A simple and fast function with very low computation costs is needed to achieve this pre-sorting. E.g., for customer records, it might be useful to consider only customers with a coincident first letter of the surname as match candidates. This obviously reduces the amount of considerable candidates, but also includes the possibility of loosing valid matchings, if, e.g., a spelling mistake appears exactly in the data used for blocking.

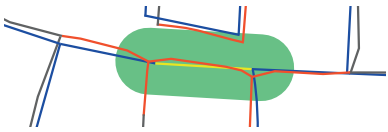


Figure 4: Match candidates are generated within a fixed-size buffer.

For road networks, we can use the spatial position of objects to generate match candidates only in the vicinity of a given object. Even for sources with a high level of generalization or inaccurate spatial exactness, it is very unlikely to find valid matchings for objects exceeding a certain distance to each other. We call this the *locality* of matchings. A **buffer** is a fixed-size area around a given object. All other objects that are completely contained inside this area or at least intersect its extent are considered as relevant vicinity of a given object. In figure 4, the green region around the currently evaluated yellow line represents the buffer area. Only those roads from the other dataset which are intersecting the buffer area (red lines) are chosen as match candidates and kept for further computations. To support adaptability, the buffer size can be set with respect to the input data. Section 4 explains heuristics to adjust the buffer size to given inputs. A fixed size buffer (independent from the cardinality of input data) helps to achieve a small and most notably *constant* number of match candidates per input object which is a crucial condition for scalability.

Check Constraints

Wagstaff et al. [WCRS01] introduce *constraints* to enhance clustering algorithms: On the one hand, *cannot-link constraints* specify that two instances must not be placed in the same cluster; on the other hand, *must-link constraints* specify that two instances have to be in the same cluster. We adapt this basic idea and both groups of constraints. Constraints improve the overall matching results and reduce the number of match candidates even further. *Must-match* and *cannot-match* constraints are used to classify matchings obligatorily as confirmed or rejected. Relying exclusively on similarity measures when comparing objects can cause problems (even if a combination of different similarity measures respects multiple characteristics): A match candidate might still obtain a high similarity value, even if one or more measures return complete dissimilarity. Imagine real-world situations where field tracks run in parallel to a highway. Even though object classes and street names do not match at all, geometric similarity (e.g., length, angle, etc.) can be high enough to erroneously match a field track and the highway. Thus it is desirable to formulate stronger conditions for matchings that can rule out similarity values.

We mainly use *rule-based* cannot-match constraints. Incorporating these rules as early as possible helps to reduce the amount of match candidates and thereby improves the overall performance. E.g., our `ObjectTypeConstraint` blacklists invalid combinations of object types and rejects candidates accordingly to these rules. Also, minimum similarity values for criteria evaluated by similarity measures can be forced by rules. E.g.:

> "Reject all pairs of objects with a corresponding line length of less than 20%."

Next, *instance-based* constraints of either type that list specific object combinations (a_i, b_j) are important for manual interaction. In preprocessing (before the actual matching algorithm starts) and postprocessing (after the algorithm stopped) these lists of object instances are queried to force and deny matchings respectively. Human operators build these lists when they correct a matching result or clarify difficult matching situations in advance to support the algorithm in its decisions.

Calculate Similarity

Matching algorithms usually rely on *similarity* between objects to identify multiple representations of the same real-world entity. Thus, the degree of similarity between two given objects needs to be expressed by a numerical value. *Similarity measures* apply a *similarity function* to given input data that returns a *similarity value*. As in, e.g., [Chr12] and [CMZ09], we calculate a numerical similarity value $s = sim(a,b)$ for objects or attributes a and b with a similarity function in a *metric space*. General requirements for these metrics may include symmetry ($sim(a,b) = sim(b,a)$) or other conditions, if needed [CMZ09]. Similarity functions should be normalized (i.e., $s \in [0,1]$) before combining multiple measures. For standard data like strings, various techniques from letter-wise or common substring comparison to phonetic similarity have been introduced [Chr12].

For our algorithm, we need to handle more complex data and introduce more sophisticated *attribute-based* and *relational* similarity measures to compare objects. The former take one single aspect of an object into account. For geodatabases we can evaluate either *semantic* attributes like, e.g., street names (strings) and speed limits (numeric values), or

the *geometric* attribute. We can derive several aspects from a geometry attribute, e.g., spatial position, shape, size, etc. which might all be used by specific measures. Relational measures highly depend on relationships between entities, i.e., context information available in a domain. In [BG07] relational similarity enhances a clustering algorithm to match bibliographic data based on the following idea: If two representations of a book, each with an associated author, match, it is easier to match other persons in a co-author-relationship for the given books. For geographic data, the topology of datasets (i.e., the underlying graph structures of road networks) can be used to inspect topological relations between spatial objects.

Instead of relying on a single measure to compute a similarity value for two given objects, we use a combination of multiple measures. A *weighted sum* combines multiple geometric and semantic attribute-based as well as relational measures *sm*:

$$sim(a,b) = \sum \omega_i * sm_i(a,b),\ \sum \omega_i = 1$$

It is important to find a combination of measures covering multiple aspects of spatial objects. I.e., at least one measure each should take care of spatial position, shape, semantic properties (if available), and context information. Some of these aspects are correlated, e.g., the line length of course depends on the shape of an object. Though, our experience has shown that even a combination of correlated similarity measures can help to improve similarity computations. Next, we introduce specific similarity measures from our library of measures:

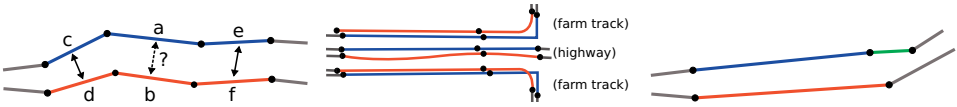
LengthSimilarity: Using the geometry of every feature we compute a numerical value for the *line length*. We relate the shorter line length ($length_{inf}$) to the longer one ($length_{sup}$) to compute a normalized similarity value: $sim_{length}(a,b) = {length_{inf}}/{length_{sup}}$.

AngleSumSimilarity: Following a single line geometry from start to end we survey the change in direction at each intermediate point. Angles α_i between incident line segments are summarized in a normalized manner ($\pi - \alpha_i$). Clockwise and counter-clockwise turns with positive and negative values that might neutralize each other are honored. Comparing the sum of all α_i for two line objects as above returns the desired similarity value.

StreetNameSimilarity: Street names are commonly used semantic attributes for road network data. Names are usually stored as simple string values. Thus, generic string similarity measures can be applied. E.g., we apply the well-known Levenshtein distance [Chr12] with fixed costs of 1 for each edit operation. Similarity values are normalized to the interval $[0,1]$ with respect to the longer string length: $sim_{streetname}(a,b) = 1 - \left({levenshtein(a,b)}/{maxLength(a,b)}\right)$. Enhancements are added by resolving abbreviations commonly used in street names, i.e., *st./street*, *rd./road*, etc.

DirectNeighbourhoodSimilarity: We propose a *relational* measure that considers already confirmed matchings in the direct vicinity of a given match candidate to evaluate its similarity. The general idea is that similarity rises, the more directly connected matchings (*neighbours*) have been already confirmed as valid matchings. This is also justified by the *principle of continuity* [HGM02], used in data visualization. It suggests that continuous shapes are usually recognized as a unit. In the sense of creating a continuous, closed line of confirmed matchings, a possible matching in between two already confirmed matchings should be confirmed (figure 5a). Additionally, situations with multiple possible match-

ings can be caused by (parallel) road segments with similar appearance (figure 5b). These situations can usually be solved if neighbouring roads are inspected first and the critical situation is matched afterwards. Finally, relational similarity can help to close small gaps (figure 5c) to create a continuous network of matchings from a global perspective.



(a) Solving critical situations with low attribute similairty.

(b) Solving ambiguous situations.

(c) Closing gaps in the graph structure.

Figure 5: Matching situations showing the usefulness of our relational similarity measure.

We can gather topological context information by exploiting the road network's underlying planar graph structure. Incident edges represent neighbouring line objects that share a common starting or end point. Previously confirmed matchings are available due to the iterative nature of our algorithm. We calculate the similarity value by relating confirmed neighbouring matches to all (i.e., confirmed and possible) neighbouring matches.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
b_1			0.95							
b_2										
b_3					0.92				0.23	
b_4			0.4				0.8			
b_5	0.1									

Figure 6: A sparse 5×10 matrix with similarity values for objects from datasets A and B.

We store all possible matchings and their similarity values in a data structure called *similarity matrix*. Though, the number of possible matchings for each input object is small due to the heavy use of blocking techniques and constraints for optimization purposes. Thus, the complete $n \times m$ matrix is a rather sparse matrix as shown in figure 6: Here, only 6 out of 50 matrix elements contain similarity values, while the rest remains empty (grey cells). Sparsity increases even more, the larger n and m grow. We have chosen a priority queue to implement the "similarity matrix" to guarantee fast access to the currently best matching: Only relevant matrix entries – possible matchings with their attached similarity value – are stored, always accessible from high to low similarity. Empty entries are omitted (figure 7). Thus, the priority queue optimizes our algorithm in terms of space and time.

0.95	0.92	0.8	0.4	0.23	0.1
(a_3, b_1)	(a_5, b_3)	(a_7, b_4)	(a_2, b_4)	(a_9, b_3)	(a_1, b_5)

Figure 7: A priority queue stores matchings and their similarity values.

Implementing the Greedy Paradigm

The SimMatching algorithm now enters its iterative part which is pictured as a loop in figure 2. In every iteration, the currently best matching, i.e., the matching with the highest similarity value, is selected from the priority queue. Here, the algorithm implements the *greedy paradigm* of making the locally optimal choice at each stage. Difficult decisions are postponed for later. For matching tasks it is not critical to find not always the global optimum. Thus, this approach promises the best trade-off between runtime and result quality. The ordered storing of matchings in our similarity matrix (efficiently implemented as priority queue) supports the greedy strategy.

Build Aggregations

With our preprocessing step and especially the line segmentation algorithm, we try to unify geometric object modeling as much as possible. However, there are lots of remaining situations where *1:1*-matchings do not reflect the real-world situation. These situations require a special form of many-to-many-matchings. Imagine a longer road that is intersected by two smaller crossroads in one dataset which results in three segments after preprocessing. If these crossroads are missing in the other dataset, e.g., because of a lack of completeness, there is no reason for a segmentation. A *3:1*-matching is required to reflect the real-world situation. We introduce *n:m*-matchings by allowing *aggregations* of objects.

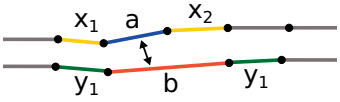


Figure 8: Possible aggregations for matching (a,b).

All neighbouring objects x_i form possible aggregation candidates $a+x_i$ for a given object a. An aggregation $a+x_i$ is considered valuable and added to the similarity matrix, if the similarity value of matching $(a+x_i,b)$ exceeds the former similarity value of matching (a,b). Otherwise, it is immediately rejected. In the next iteration of our algorithm, this step is repeated. We now consider matchings $(a+x_i+x_j,b)$ and $(a+x_i,b+y_i)$ for neighbouring objects x_j and y_i. Here, we follow the greedy paradigm again: Only the best matching and then the best aggregation are considered. If this step is repeated while similarity increases, aggregated objects can grow to n or m parts, which build a new object. The basic algorithm requires no changes as we handle the resulting *n:m*-matchings as *1:1*-matchings between aggregations.

Apply Matching Rules

In the final step of each iteration, *Matching rules* classify all possible matchings – one matching per iteration. The greedy strategy suggests to classify the currently most confident choice first. In each iteration, only the currently best matching, i.e., the top of the similarity matrix, is either confirmed or rejected. Matching rules usually base this decision on similarity values and other knowledge gathered in previous steps. Though, they may also incorporate new information that is considered by the rules. Combined with all

implicit rules set by constraints, simple threshold rules provide excellent results. E.g.: "Possible matchings are confirmed while their similarity value is higher than 0.75". The threshold depends on input data, selected measures, and their weights (see section 4).

We perform various *clean up* operations for the similarity matrix, each time a matching (a,b) is confirmed: The matching itself and all possible matchings including a or b are removed from the matrix, thus implicitly rejected. We furthermore remove aggregated objects built from a or b, or parts of a or b if either are aggregations themselves. These operations avoid conflicts in the final matching result which could appear if object a is part of multiple confirmed matchings. They also help to decrease the size of the similarity matrix with every classification step.

Classification continues while more matchings can be confirmed. The algorithm exits its iterative part and ends if there are no more matchings that fulfill the given matching rules. I.e., for threshold rules, all remaining possible matchings have a similarity value lower than the given threshold. The set of confirmed matchings together with two sets of unmatched objects – one for each input source – are called *matching result*. Due to previous clean up operations performed on the similarity matrix the result is always unambiguous and conflict-free: No object can be part of multiple matchings, multiple aggregations, matching and aggregation, or matched and unmatched at the same time.

Recalculate Similarity

By confirming more and more matchings based on their good attribute similarity, we can now start to take advantage of context information. Neighbouring objects profit from previously confirmed matchings as their relational similarity increases. In every iteration, we have to update the similarity matrix to the new context situation by locally *recalculating similarity*, i.e., updating relational similarity. Though, we do not recompute the whole matrix: Updates are needed only for the direct vicinity of the confirmed matching and are limited to all neighbouring objects of the confirmed matching.

Partitioning and Parallel Execution

Within our previous work [ML04, TL07] we experienced very long computation times, if matching computations – especially geometric computations – were executed directly inside a database management system with spatial extension. We therefore have designed our SimMatching approach as an *in-memory algorithm* running on standard client machines with dedicated geometry libraries, but also with restricted hardware resources. The amount of data that can be processed at once is limited by the size of the available main memory. To process larger input data, we rely on a *partitioning* framework [WL12] that splits larger input data into manageable pieces called *partitions*. Partitioning is justified by the locality of matchings as explained before. Runtime and memory consumption depend more on the number of processed objects (the cardinality of the input datasets) than on the size of the processed area. Thus, we propose an *adaptable partitioning system* that adapts the number and size of partitions to the number of contained objects.

All objects inside the extent of a certain partition are queried from a spatial database system by a spatial range query supported by a spatial index. After applying our matching process

in-memory, a *recomposition* phase combines the results for multiple partitions. Conflicts, especially near partition borders, have to be resolved before writing the results back into the database. We expand every partition with a buffer operation to create a soft border that contains context information for relational similarity measures. Besides this overlapping, partitions remain disjoint which allows us to process them independently in parallel. *Parallel execution* on multi-core systems leads to astonishing performance enhancements and decreases execution time proportionally to the number of used threads.

4 Results

In this section, we apply our SimMatching approach to various real-world data sources covering parts of Germany and the United States. We analyze these input sources with their characteristics and describe our testing configuration. Result quality is evaluated with quality measures from the information retrieval domain. Performance tests prove efficiency and scalability of our approach and allow a comparison to existing systems.

Real-world Input Data Description

ATKIS: The *Authoritative Topographic-Cartographic Information System* is established and administered by a federal German institution. It provides a semantically and geometrically rich large scale landscape model and is very well documented. An *object catalog* lists all available object classes together with their specific attributes and well-defined object modeling rules. Exactness of spatial positions is very important for this dataset.

OSM: *OpenStreetMap* is a collaborative open data project to create a free worldwide map with the help of a global community. OSM data can be geometrically precise and semantically rich if contributors follow the project guidelines. Though, data quality varies as both novices and experts contribute. A high number of volunteers frequently updates OSM data for high currentness. Owners of proprietary datasets might be interested to utilize this currentness for improving their data quality by matching.

D20: This proprietary dataset supplies cartographically generalized road networks for producing printed maps. Objects are shaped manually by visual aspects and do not follow a strict modeling scheme. Spatial exactness is needed only so far that signatures appear correctly. Besides street names and object classes, semantic richness is low.

NAV: Created for routing purposes, this proprietary dataset is mainly used in car navigation systems. Hence it includes only car-accessible roads that were highly cartographically generalized for on-screen presentation. Attributes specialized for driver's needs include, e.g., speed limits, turning restrictions, and many others, but just coarse object classes.

TIGER: Road network data for the United States are supplied by the US Census Bureau and released into the public domain. Data are of high currentness (yearly updates) and attributed with administrative information, e.g., ZIP codes, and address ranges. TIGER data was formerly used as base for US OSM data, but differs now after edits in both sources.

Heuristics for parameter optimization

Parameters for our algorithm can be customized for every matching task and thereby support adaptability and efficiency of the algorithm. E.g., matching OSM with ATKIS data requires different parameters than matching OSM and D20 data due to the diversity of the input sources. Configurable parameters are the buffer size for candidate generation, the choice and the weights of similarity measures, and the threshold for classifying matchings. Finding appropriate parameter values starts with a detailed analysis of the input sources and their characteristics. The buffer size depends on the exactness of spatial positions in the input data and the resulting distance between different road representations. A higher degree of cartographic generalization (e.g., see figure 1a) requires larger buffers than more comparable sources (e.g., see figure 1b). Measuring average distances on a sample basis and adding few meters tolerance is an appropriate choice. The buffer size influences efficiency which depends on a suitable amount of match candidates. Too large buffers result in a too high number of candidates – too small buffers cause missing matchings.

Weights for similarity measures depend on the availability, significance, and quality of attributes. Geometric measures tend to be more unreliable with stronger generalization. Street names might be incomplete in many datasets and are then not usable for similarity computations. Measures based on object types should be weighted lower, if there are just few and coarse object types available, like in our NAV dataset. These heuristics give an evidence of a "good" choice of measures and weights. They should be empirically verified and further optimized by test runs with subsequent plausibility checks. Finally, the threshold can be approximated by manually classifying matchings repeatedly as confirmed or rejected. After doing this on a sample basis, the threshold converges to a value lower/higher than all similarity values of accepted/rejected matchings.

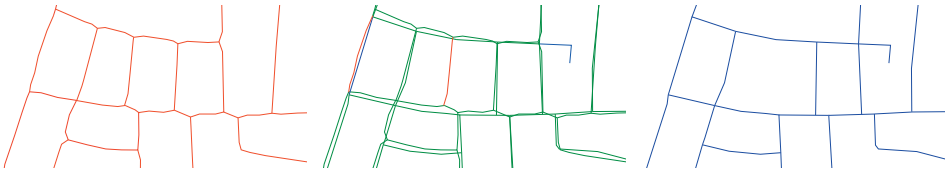
Testing Configuration

The SimMatching algorithm is implemented in *Java 7*. *OpenJUMP*, an open source Geographic Information System (GIS), is used to visualize input data and matching results. It provides an extensive class library for spatial data types (based on the *JTS Topology Suite*). On the server side we use a *PostgreSQL 9.3* database with the spatial *PostGIS* extension and its geometry data types and functions. Java thread pooling helps to prepare tasks for each partition, schedule their execution and merge threads for result collection. We also use pooled database connections to allow one exclusive connection for each active thread. Of course, (spatial) indexes are used extensively, particularly to speed up range queries to select all features inside a given partition.

We chose D20 and OSM as input sources with high diversity for testing purposes. Parameters were optimized following the heuristics introduced in the former section. We achieved best results for matching D20 and OSM data when using the following measures with their respective weights and a threshold of 0.55: `LengthSimilarity` (0.3), `AngleSumSimilarity` (0.1), `AngleSimilarity` (0.1), `Starting-` and `End-PointSimilarity` (0.05 each), `StreetNameSimilarity` (0.1), and `Direct-NeighbourhoodSimilarity` (0.3). All tests were performed on regular desktop hardware based on a year 2013 quad core CPU.

Result Quality Evaluation

We evaluated the quality of matching results created by our SimMatching approach by inspecting a visualization of matchings, unmatched objects, and original input data for either source. Figure 9b shows a small map excerpt containing matchings (green) and unmatched objects – red objects originating from the dataset pictured in figure 9a and blue objects originating from the other (figure 9c). Most objects are matched correctly. There are correct unmatched objects as well: The single red and blue objects miss a respective counterpart from the other dataset and therefore shall remain unmatched. However, on the far left side (figure 9b) two roads showing a high geometric similarity are also unmatched. After inspecting other attribute data not pictured here, we have to decide if the algorithm failed here and classify the matching possibly as missing, i.e., not found by our algorithm.



(a) Input data "red". (b) Matchings (green) and unmatched objects (red and blue). (c) Input data "blue".

Figure 9: Visualizing input data and a matching result.

For a sophisticated evaluation of result quality, we manually analyzed a map excerpt around *Munich*, Germany, for OSM and D20 data. Our dataset *Munich* (figure 10) covers an area of approximately 7.2 km^2. There are 1 065 OSM and 1 104 D20 objects contained in the original input data which resulted in 2 920 OSM and 1 114 D20 objects after preprocessing. Using a process configuration as described in section 4, a total of 969 matchings were confirmed and the execution took below 4 seconds.

We applied measures known from the information retrieval domain, *precision* and *recall*, to have a comparable quality measure. Precision indicates the rate of correctly classified confirmed matchings in all found matchings. We identified 32 situations where wrong matchings were confirmed or a better solutions (e.g., a larger aggregation) would have been more suitable. In relation to a total of 969 matchings, we reached a precision value of 97%. Next, recall describes the rate of correctly classified confirmed matchings found in relation to all valid matchings. Considering missing and non-optimum matchings, we determined a recall value of 97% as well. In a previous evaluation [SL14], both values evaluated to 95% each. An enhanced preprocessing step with multi-lane handling as well as tuned measures and constraints, slighly improved these values.

Performance Evaluation

For a comprehensive performance evaluation we have matched several map excerpts of increasing size for different regions of Germany (figure 11). The number of objects is determined by the cardinality of the larger database (here: OSM) before preprocessing.

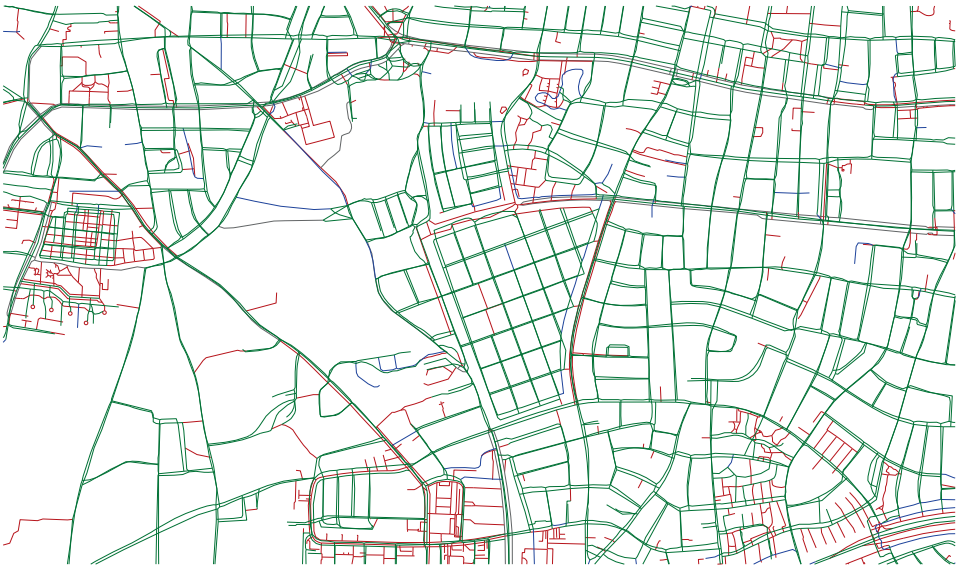


Figure 10: Matchings (green), unmatched OSM objects (red), and unmatched D20 objects (blue) for testing area *Munich*.

Total runtimes include partitioning, loading data from a spatial database, preprocessing, the actual matching step, and finally recomposing and writing back the results. All given times are average values generated from 20 independent test runs per dataset.

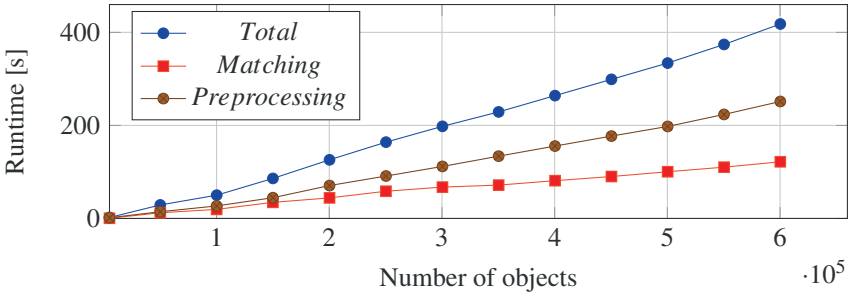


Figure 11: Runtimes for input data of increasing size.

While we see total runtimes below one minute for 100 000 input objects, they increase to below three and a half minutes for 300 000 objects. The largest dataset of 600 000 objects took approximately seven minutes to be fully processed. All in all, we assume a linear runtime for our approach based on these experiments. Therefore we have proven excellent scalability to very large geodatabases. Likewise, the two listed subprocesses show a linear time complexity, too. Preprocessing takes almost twice the time of the actual matching as more geometry-changing operations are performed. This step also includes more database

communication as preprocessed data has to be written to the spatial database system and is organized there, i.e., by enabling database constraints and (spatial) indexes.

Performance Comparison

Directly comparing the results of our performance tests with other approaches is usually not feasible as most input data is distributed under proprietary licences and not available to be tested with our system. Next, the cardinality, dimension, and complexity of used testing areas is not always clear. We also have to consider the evolution of hardware resources when comparing runtimes. However, we try to respect test results and complexity analysis whenever they are available. We propose the unit *objects per second* for measuring how many input objects can be processed in one second by the respective algorithm.

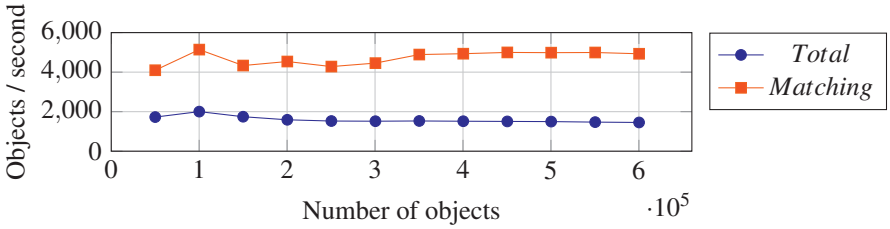


Figure 12: Processed objects per second for input data of increasing size.

The SimMatching approach is able to process between 1 450 and 2 000 objects per second based on the total computation time (figure 12). For the actual matching only, the values increase to 4 100 to 5 100 objects per second (figure 12). There are minor variations for the different datasets as many factors, that cannot be easily unified, influence the algorithm performance. Next to the cardinality this includes, e.g., the network complexity. On average, values of 1 650 and 4 700 (matching only) objects per second are reached.

The original buffer growing algorithm [WF99] was used to process a tiny test area of $2km \times 2km$. The authors expected an exponential time complexity. Even for the small map excerpt containing only 963 objects, the computation time was almost as high as 7 hours. This evaluates to a performance value of 0.2 objects per second. For the enhanced buffer growing [ML04] developed by our group, internal technical reports show an increase to 1.2 objects per second. The graph matching [TL07] is able to process about 10 objects per second for a dataset of 100 000 objects in a map excerpt of $40km \times 40km$.

In 2009, Zhang [Zha09] tested his approach on German datasets of similar size like in figure 11. He assumed a linear time complexity – dependent on the number of input objects. His approach was experimentally proven to be scalable to large datasets. A matching performance of 500 objects per second is mentioned in [Zha09]. The *ApproxMatching* algorithm proposed in [SKSD13] for ad hoc matching was optimized for performance. With their fastest variant (not necessarily bringing the best recall and precision values), the authors were able to match 18 264 objects in about 110 seconds. This evaluates to a total of 166 objects per second. However, it is not clear if, e.g., the creation time for used indexes or communication costs were included in these values. Next, the input data (Manhattan,

similar to figure 1b) seemed to be of low structural complexity. The authors concluded an "almost linear" runtime, but did not test any larger datasets than the one mentioned above.

All in all, we can conclude that our SimMatching approach outperforms selected existing approaches in terms of matching performance. Even algorithms that are optimized for processing input data in short time [Zha09, SKSD13] and that are estimated to have a linear time complexity with respect to the cardinality of input data, perform worse in terms of processed objects per second.

5 Conclusion and Future Work

In this paper, we have presented an adaptable, efficient and scalable solution to the challenging task of integrating diverse, complex and large geodatabases. Our SimMatching approach provides an end-to-end-system based on a simple algorithmic structure with flexible components and configurable parameters. By choosing appropriate similarity measures and respective weights, the algorithm adapts to arbitrary input data. A combination of geometric, semantic and relational measures produces high quality results in terms of recall and precision when applying the algorithm to several real-world data sources. In each iteration, previously confirmed matchings and context information from the underlying graph structure of road networks help to improve matching decisions in critical situations. Though, without any negative performance impact seen elsewhere when graph handling becomes dominant.

We extensively use blocking with spatial buffers and other techniques to keep the number of candidates low. This guarantees short runtimes and an excellent matching performance. This is further supported by a greedy strategy to process locally best solutions first. We have shown that our algorithm outperforms other approaches in terms of processed objects per second while result quality stays competitive. This efficiency together with a partitioning framework wrapped around the actual matching algorithm leads to a high scalability to large datasets. Tests with input data of increasing cardinality have shown that the algorithm scales very well. We habe experimentally verified a linear runtime complexity.

For future work, we have to validate the linear runtime observed in our experiments with an in-depth theoretical complexity analysis. Furthermore, we plan to employ and improve machine learning techniques for configuring our approach. Currently, parameters are manually optimized: Appropriate combinations of similarity measures have to be chosen and other parameters have to be adjusted dependent on the input data. This is a time-consuming task, which is done by a domain expert. First tests with Support Vector Machines (SVM) have been promising and have shown that weights are converging when enough sample data is supplied by a human operator within an active learning system.

References

[BG07] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):Article 5, 2007.

[Chr12] P. Christen. *Data Matching*. Springer, 2012.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 3rd edition, 2009.

[CMZ09] S. Chen, B. Ma, and K. Zhang. On the similarity metric and the distance metric. *Theoretical Computer Science*, 410(24-25):2365–2376, 2009.

[Con97] S. Conrad. *Föderierte Datenbanksysteme*. Springer, 1997.

[DFE01] Y. Doytsher, S. Filin, and E. Ezra. Transformation of datasets in a linear-based map conflation framework. *Surveying and Land Information Systems*, 61(3):165–176, 2001.

[DHI12] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier, 2012.

[HGM02] G. Hake, D. Grünreich, and L. Meng. *Kartographie*. De Gruyter, 2002.

[LN07] U. Leser and F. Naumann. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. Dpunkt.Verlag GmbH, 2007.

[MGR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, 2002.

[ML04] D. Mantel and U. W. Lipeck. Matching Cartographic Objects in Spatial Databases. In *Proceedings of the 20th Congress of the ISPRS*, pages 172–176, 2004.

[RB01] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[Saa88] A. Saalfeld. Conflation – Automated map compilation. *International Journal of Geographical Information Systems*, 2(3):217–228, 1988.

[SKSD13] E. Safra, Y. Kanza, Y. Sagiv, and Y. Doytsher. Ad hoc matching of vectorial road networks. *Int. Journal of Geographical Information Science*, 27(1):114–153, 2013.

[SL14] M. Schäfers and U. W. Lipeck. Spatial Data Integration Using Similarity-based Matching. In H.-M. Haav, A. Kalja, and T. Robal, editors, *Proc. of the 11th Int. Baltic Conference on Databases and Information Systems*, pages 89–100. TUT Press, 2014.

[TL07] M. Tiedge and U. W. Lipeck. Graphbasiertes Matching in Räumlichen Datenbanken. In *Proc. of the 19th GI-Workshop Grundlagen von Datenbanken*, pages 42–46, 2007.

[WCRS01] K. Wagstaff, C. Cardie, S. Rogers, and S. Schroedl. Constrained K-means Clustering with Background Knowledge. In *Proceedings of the 18th International Conference on Machine Learning*, pages 577–584, 2001.

[WF99] V. Walter and D. Fritsch. Matching spatial data sets: a statistical approach. *International Journal of Geographical Information Science*, 13(5):445–473, 1999.

[WL12] H. Warneke and U. W. Lipeck. Ein Partitionierungsdienst für Geographische Daten in Räumlichen Datenbanken. In *Proceedings of the 24th GI-Workshop Grundlagen von Datenbanken*, pages 35–40, 2012.

[Zha09] M. Zhang. *Methods and Implementations of Road-Network Matching*. PhD Thesis, Technische Universität München, 2009.

Relationships for Dynamic Data Types in RSQL

Tobias Jäkel*, Thomas Kühn+, Stefan Hinkel*, Hannes Voigt* and Wolfgang Lehner*

* Database Technology Group
{tobias.jaekel, stefan.hinkel, hannes.voigt, wolfgang.lehner}@tu-dresden.de
+ Software Technology Group
thomas.kuehn3@tu-dresden.de
Technische Universität Dresden
Nöthnitzer Str. 46
01187 Dresden

Abstract: Currently, there is a mismatch between the conceptual model of an information system and its implementation in a database management system (DBMS). Most of the conceptual modeling languages relate their conceptual entities with relationships, but relational database management systems solely rely on the notion of relations to model both, entities and relationships. To make things worse, real world objects are not static as assumed in such modeling languages, but change over time. Thus, modeling languages were enriched to model those scenarios, as well. However, mapping these models onto relational databases requires the use of object-relational mapping engines, which in turn hide the semantics of the conceptual model from the DBMS. Consequently, traditional relational database systems cannot directly ensure specific consistency constraints and thus lose their meaning as single point of truth for highly distributed information systems. To overcome these issues we have proposed RSQL, a data model and query language introducing role-based data structures in DBMSs. Despite the fact that RSQL is able to handle complex objects, it does not support relationships between those objects. Therefore, this work adds relationships to RSQL by augmenting the data model and extending its query language. As a result, this extension allows for the direct representation of conceptual models with complex objects and relationships in the DBMS. Thus, relationships can be directly addressed in queries and the DBMS automatically ensures relationship consistency constraints as well as cardinality. In sum, a DBMS equipped with the extended RSQL is apt for storing and querying conceptual models and thus regains its rightful position as the single point of truth for highly distributed information systems.

1 Introduction

Currently, there is a mismatch between the conceptual model of an information system and its representation in a *Database Management System* (DBMS). For instance, if the conceptual model is mapped to a relational database schema, this results in different views on concepts specified on the conceptual level. Most of the widely used conceptual modeling languages are based on objects and relationships to relate these objects, like ER [Che76] or UML [RJB10]. However, relational databases only rely on the notions of re-

lations to model both objects and relationships. To make things worse, real world objects and their relationships can change over time and usually do. Hence, conceptual modeling languages were enriched by features, like the role concept, to allow for the specification of more complex and dynamic systems. Based on this observation, Bachman has proposed the role concept in 1977 [BD77], where the core of an object is separated from its relationship dependent parts, namely roles. This enables information system designers to model dynamics of objects on the conceptual level by representing all relationship dependent attributes and behavior in several roles. Hence, role-based modeling reduces the complexity and allows for continuous evolution of highly distributed information systems. For instance, imagine a person who can be a consultant and a customer. In object-oriented modeling one would define subclasses using inheritance for each combination, which leads to an exponential explosion of subclasses and to reinstantiation each time the person changes the subclass at run-time[Ste99]. Contrariwise, using role-based modeling would only describe the core object and its extension that can be added and removed dynamically during run-time. Thus, subclass explosion and run-time reinstantiation are avoided. However, the semantics of the role concept is completely unknown to relational DBMSs, thus, the gap between conceptual models and their representation in the database becomes even bigger.

This becomes a problem for highly distributed information systems that rely on a DBMS as **single point of truth**. Because data objects are shared among several subsystems, only the DBMS persists data and ensures their consistency for all applications and users. Hence, the DBMS has to provide global consistency, whereas the subsystems have their local perspective. However, current DBMS are not apt for this task, because they lack the notion of complex objects and relationships. Thus, most of the semantics introduced in the conceptual model is lost during the persisting and transformation process. Consequently, most of the semantics and its validation is implemented in mapping engines running in the various subsystems. Nonetheless, they cannot ensure the consistency of the whole distributed information system. Hence, relational DBMSs must be extended to represent roles and relationships directly to regain their characteristics as **single point of truth**.

One step in this direction was the introduction of RSQL in [JKVL14], which extends a classical DBMS to represent dynamic, complex data structures with roles. This definition comprises dynamic data structures and a query language, but does not consider relationships as first-class citizens. Unfortunately, traditional relational methods are insufficient for directly representing relationships. Data in relational DBMS are stored based on relational semantics and thus relationships must either be mixed into relations of other concepts or mapped to a separate relation. In the former case, the relationship attributes and references become intermingled with relations representing concepts. Whereas in the latter case, the separate relation cannot be distinguished from concept relations within the DBMS. Moreover, both approaches lead to fragile schemas, since every cardinality change on the conceptual level causes extensive database schema changes. In sum, we extend RSQL by first-class relationships that preserve the semantics of the conceptual model.

Hence, this paper has the following contributions. Firstly, we present a formal definition for relationships as an extension to Dynamic Data Types. Secondly, we extend RSQL's *Data Definition Language* (DLL), *Data Manipulation Language* (DML), and *Data Query Language* (DQL) to directly represent relationships. Hence, we introduce new database

objects on the type as well as on the instance level. Furthermore, we argue that the direct representation of relationships makes the DBMS more robust against cardinality changes. These contributions ensure a direct representation of role-based objects and relationships in relational DBMS. Finally, DBMS regain the ability to be the single point of truth for highly distributed role-based information systems by providing and ensuring consistent persistence of dynamic complex data objects and their relationships.

This paper is structured as follows: In the following section a running example is introduced. Afterwards, Section 3 presents an extended formal data model. This is followed by the RSQL extension in Section 4, its classification and related work in Section 5. Finally, conclusions and future work are discussed in Section 6.

2 Running Example

The most important feature of the role concept for modeling is to express dynamics in typecasting of objects. Basically, this concept distinguishes between *Natural Types* and *Role Types*. Natural Types represent the core and immutable type of an object that cannot be discarded without ceasing to exist. In contrast, Role Types represent the mutable type of an object, which can be acquired and abandoned dynamically. Over time Naturals, the instances of Natural Types, may start or stop playing several Roles. Thus, they can acquire additional attributes dynamically. To constrain which Natural Types *can play* which Role Types they are restricted by a fills-relation.

A Natural Type and its connected Role Types form a *Dynamic Data Type* (DDT) that is handled as individual type by the DBMS. Thus, DDTs are the main data structure in a role-based DBMS. However, Role Types are not exclusive to a single DDT, rather they can be shared between several Dynamic Data Types. Furthermore, a DDT restricts the types of roles that can be assigned to an instance of that DDT. This definition enables DBMS to guarantee role-specific consistency conditions. Additionally, DDTs can only be related by Relationship Types between Role Types. This enables relationships independently of the DDT, since Role Types can be part of several DDTs.

Figure 1 illustrates a small banking application, that serves as running example. It describes *Customers* that are related to *CheckingsAccounts*. Between **Accounts** money can be transferred and a *Customer* can have one or more *Consultants*. The example consists of three DDTs: **Person**, **Company** and **Account**. Both, **Person** and **Company** can be a *Customer*, but only **Person** is allowed to be a *Consultant* as well. *Consultant* and *Customer* are related by a Relationship Type *advices*. The third DDT describes different facets of **Accounts**. An **Account** can be a *CheckingsAccount* and in this role it has to be related to exactly one *Customer*. Additionally, **Accounts** can play the roles of the Role Type *Source* and *Target* to describe transactions of transferring money between **Accounts**. This transaction is represented by the *transfer* Relationship Type. Moreover, a unique *Target* counterpart for each *Source* has to exist.

One possible instance of this model is shown in Figure 2. It comprises two **Persons** *Peter* and *Klaus*, as well as a **Company** *Google*, whereas *Peter* plays the role of a *Consultant*

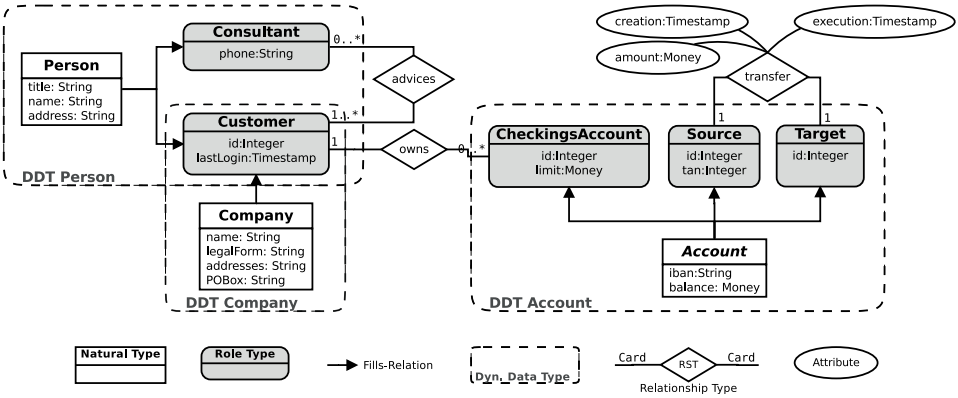


Figure 1: Role Modeling Example - Customers, Bank Accounts, and Transactions

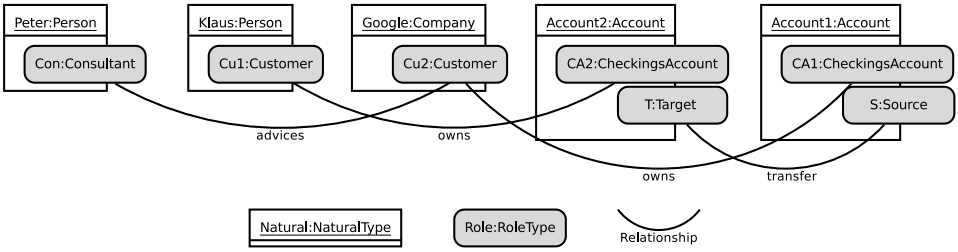


Figure 2: Instance of the Role Modeling Example (Fig. 1)

and the other two are *Customers*. Each of these customers *owns* one CheckingsAccounts, however, only *Google* is *advised* by *Peter*. Additionally, the model contains one transaction from *Account2* to *Account1*, which play the roles *Source* and *Target*, respectively, via the *transfer* relationship. To indicate, which Natural plays which Role, the Roles are placed at the border of their respective players. Throughout this paper, both the role model and its instance are used as a running example.

3 Dynamic Data Types and Relationship Types

This section introduces the notion of Relationship Types to represent dynamic relationships. To do this, we henceforth, present a formal model for both the type and the instance level. In particular, this definition is based on the formal model of *Dynamic Data Types* (DDT) presented in [JKVL14]. Consequently, we augment this model by including Relationship Types and the corresponding semantics. To be coherent, we present complete definitions and examples for both levels, but focus on Relationship Types in the discussion.

3.1 Type Level

In its core, the type level definition is comparable to Friedrich Steimann's type definition for *LODWICK* in [Ste99]. Dynamic Data Types consist of Natural Types and Role Types [JKVL14]. Roughly, Natural Types are rigid and non-founded. That means an instance of a Natural Type, denoted as Natural, loses its identity by changing the type. Naturals can exist independently of relations to other individuals in the systems. Role Types are exactly the opposite of Natural Types, they are non-rigid and founded, whereas the latter indicates their dependence on other individuals [Ste99, Gua92]. Consequently, Relationship Types are defined between two Role Types representing their two ends. For reason of simplicity and in contrast to *LODWICK* [Ste99], we focus solely on binary relationships with a limited set of cardinality constraints.

Definition 1 (Schema)**.** *Let NT be the set of all Natural Types, RT the set of all Role Types, RST be the set of Relationship Types, and $Card = \{0..1,\ 1..1,\ 1..*,\ 0..*\}$ be the set of available cardinalities with $NT \cap RT = \emptyset$. The schema s is then given by the tuple $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$, where $\mathit{fills} \subseteq NT \times RT$ is a relation for Natural Types fulfilling Role Types and $\mathit{rel} : RST \to RT \times Card \times Card \times RT$ is a function mapping a relationship type to the two Role Types at its end and to the corresponding cardinalities. Because Role Types cannot exist on their own and Relationships are prohibited to link the same Role Type, we constrain the schema s by:*

$$\forall rt \in RT\ \exists nt \in NT\ .\ (nt, rt) \in \mathit{fills} \tag{1}$$

$$\forall \mathit{rst} \in RST\ \nexists rt \in RT\ .\ \mathit{rel}(\mathit{rst}) = (rt, c_1, c_2, rt) \tag{2}$$

Applying these constraints ensures that (1) there exists an association of a Role Type to at least one Natural Type by using the *fills*-relation and that (2) each Relationship Type is defined between two distinct Role Types. Henceforth, we use the infix notation nt *fills* rt.

The cardinality constraints introduced here differ in their semantics from ER, where cardinality constraints directly describe the relationship between two static entity types. In our model we have to distinguish between two different relations, the *rel*-relation and the *fills*-relations. The former describes the relation between two distinct Role Types and constraints on this relation basically define how often an individual Role can be related to a counter Role. In contrast, the latter defines the relation between a Natural Type and a Role Type to form Dynamic Data Types. However, constraints on this particular relation result in constraining how often a Natural can play a Role of the same Role Type simultaneously, which is an internal Dynamic Data Type constraint. Such constraints, however, are beyond the scope of this paper. Thus, the *fills*-relation is assumed to be unconstrained in our definition. As consequence, one-to-one and one-to-many relationship constraints are only ensured between Roles and not their various players, such that traditional relationship constraints between two Natural Types cannot be enforced.

Example 1 *Let $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema for the bank application shown in Fig. 1, where $NT = \{\mathit{Person}, \mathit{Company}, \mathit{Account}\}$ is the set of Natural Types, $RT = \{\mathit{Customer}, \mathit{Consultant}, \mathit{CheckingsAccount}, \mathit{Source}, \mathit{Target}\}$ are the Role Types,*

and $RST = \{owns, advices, transfer\}$ *is the set of Relationship Types. Furthermore, the fills-relation is defined as:*

$$\begin{aligned} \textit{fills} \stackrel{\textit{def}}{=} \{&(\textit{Person}, \textit{Customer}), \quad (\textit{Person}, \textit{Consultant}), \quad (\textit{Company}, \textit{Customer}), \\ &(\textit{Account}, \textit{Source}), \quad (\textit{Account}, \textit{Target}), \quad (\textit{Account}, \textit{CheckingsAccount})\} \end{aligned}$$

and the rel-function is defined for each Relationship Type as:

$$\begin{aligned} \textit{rel}(\textit{owns}) &\stackrel{\textit{def}}{=} (\textit{Customer}, 1..1, 0..*, \textit{CheckingsAccount}) \\ \textit{rel}(\textit{advices}) &\stackrel{\textit{def}}{=} (\textit{Consultant}, 0..*, 1..*, \textit{Customer}) \\ \textit{rel}(\textit{transfer}) &\stackrel{\textit{def}}{=} (\textit{Source}, 1..1, 1..1, \textit{Target}) \end{aligned}$$

As it turns out, the $bank$ schema is a direct representation of the graphical model shown in Figure 1 such that each Role Type, Natural Type, and Relationship Type is an element of the corresponding set and each line is mapped to the *fills*-relation or the *rel*-function depending on its kind. As next step, the notion of a DDT is defined as a composition of a Natural Type nt and all Role Types it fills:

Definition 2 (Dynamic Data Type)**.** *Let* $s = (NT, RT, RST, \textit{fills}, \textit{rel})$ *be a schema and* $nt \in NT$ *be a Natural Type. A Dynamic Data Type is then defined as* $ddt = (nt, RT_{nt})$ *with* $RT_{nt} \subseteq RT$ *such that* $RT_{nt} \stackrel{\textit{def}}{=} \{rt \in RT \mid nt \textit{ fills } rt\}$

In addition to that, DDTs are considered as individual types consisting of one Natural Type in its center and several Role Types, which are related in the *fills*-relation. Besides that, each DDT gives rise to a number of Configurations in which instances of that DDT might appear. To put it differently, a Configuration specifies by which specific Role Types a DDT is extended. Thus, it is not defined by the user directly but deduced from the given *fills*-relation.

Definition 3 (Configuration)**.** *Let* $ddt = (nt, RT_{nt})$ *be a Dynamic Data Type; a Configuration of this DDT is then given by* $c = (nt, RT^c)$*, where* $RT^c \subseteq RT_{nt}$*.*

Notably, since Configurations are defined on the type level, playing multiple Roles of the same Role Type simultaneously does not affect the Configuration.

Example 2 *Applying these definitions to the schema* $bank = (NT, RT, RST, \textit{fills}, \textit{rel})$ *gives rise to the following three DDTs:*

$$\begin{aligned} ddt_{\textit{Person}} &= (\textit{Person}, \{\textit{Customer}, \textit{Consultant}\}) \\ ddt_{\textit{Company}} &= (\textit{Company}, \{\textit{Customer}\}) \\ ddt_{\textit{Account}} &= (\textit{Account}, \{\textit{CheckingsAccount}, \textit{Source}, \textit{Target}\}) \end{aligned}$$

As a side note, each of these DDTs has a distinct number of possible Configurations, e.g., ddt_{Person} has four, ddt_{Company} has two, and so on. A Configuration characterizes a possible subset of Role Types extending a given DDT. For more information on Configurations the reader can refer to [JKVL14]. In conclusion, the extended schema additionally contains a set of Relationship Types which relate two Role Types and impose cardinality constraints on these relationships.

3.2 Instance Level

In contrast to Steimann's *LODWICK* [Ste99], we do not only instantiate Natural Types to Naturals but also Role Types to Roles. Thus, we are now able to directly represent relationships between two corresponding Roles.

Definition 4 (Instance)**.** *Let $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema, N the set of all Naturals, and R the set of all Roles with $N \cap R = \emptyset$. An instance i of this schema s is then defined as $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$, where $\mathit{type} : (N \to NT) \cup (R \to RT)$ is a polymorphic function assigning a distinct Natural Type or Role Type to each Natural or Role, respectively; $\mathit{plays} \subseteq N \times R$ is a relation defining which Naturals play which Roles; and $\mathit{links} : RST \to 2^{R^\varepsilon \times R^\varepsilon}$ is a function from Relationship Types to their extend set with $R^\varepsilon = R \cup \{\varepsilon\}$ and $\varepsilon \notin R$. As shorthand notation we define two index sets for Naturals and Roles.*

$$N_{nt} \stackrel{def}{=} \{n \in N \mid \mathit{type}(n) = nt\} \qquad \text{for } nt \in NT$$

$$R_{rt} \stackrel{def}{=} \{r \in R \mid \mathit{type}(r) = rt\} \qquad \text{for } rt \in RT$$

Moreover, we require the following five axioms to hold for any instance i of the schema s:

$$\forall r \in R\, \exists!\, n \in N \,.\, n \mathit{\ plays\ } r \tag{1}$$

$$\forall (n, r) \in \mathit{plays} \,.\, \mathit{type}(n) \mathit{\ fills\ type}(r) \tag{2}$$

$$\forall \mathit{rst} \in RST \,.\, \mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \wedge \mathit{links}(\mathit{rst}) \subseteq (R^\varepsilon_{rt_1} \times R^\varepsilon_{rt_2}) \setminus \{(\varepsilon, \varepsilon)\} \tag{3}$$

$$\forall \mathit{rst} \in RST \,.\, \mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \wedge (\forall r_1 \in R_{rt_1}\, \exists (r_1, r) \in \mathit{links}(\mathit{rst})) \wedge (\forall r_2 \in R_{rt_2}\, \exists (r, r_2) \in \mathit{links}(\mathit{rst})) \tag{4}$$

$$\begin{aligned} \forall \mathit{rst} \in RST\, \forall (r_1, r_2) \in \mathit{links}(\mathit{rst}) \,.\, \mathit{rel}(\mathit{rst}) = (rt_1, c_1, c_2, rt_2) \wedge \\ (c_1 \in \{1..1, 1..*\} \Rightarrow r_1 \neq \varepsilon) \wedge \\ (c_1 \in \{0..1, 1..1\} \Rightarrow \nexists (r_1', r_2) \in \mathit{links}(\mathit{rst}) \,.\, r_1 \neq r_1') \wedge \\ (c_2 \in \{1..1, 1..*\} \Rightarrow r_2 \neq \varepsilon) \wedge \\ (c_2 \in \{0..1, 1..1\} \Rightarrow \nexists (r_1, r_2') \in \mathit{links}(\mathit{rst}) \,.\, r_2 \neq r_2') \end{aligned} \tag{5}$$

$$\forall rst \in RST\, \forall c \in C\, \forall (r_1, r_2) \in (\mathit{links}(rst) \cap R \times R) \,. (r_1, \varepsilon), (\varepsilon, r_2) \notin \mathit{links}(rst) \tag{6}$$

Both, the *plays*-relation and the *links*-function are the instance level equivalent to the *fills*-relation and the *rel*-function on the type level. The former captures which Natural plays which Role at the moment. To simplify things, we use the infix notation n *plays* r for this relation. Whereas the latter maps each Relationship Type to its extension, i.e. the set of tuples of Roles which are currently related. Consequently, each Relationship Type gives rise to several relationships represented by the aforementioned tuples.

Besides the general definition of instances of a schema, it is crucial to discuss some of the axioms required to hold for each instance, because these axioms enforce consistency with

respect to the schema. The first two axioms (1-2) ensure that the *plays*-relation assigns exactly one Natural to each Role as its player and is type conform to the *fills*-relation defined in the schema. Similarly, the next three axioms (3-6) constrain the *links*-function to ensure its consistency according to the database schema. In particular, axiom (3) ensures that *links* return only those tuples of roles for a given Relationship Type that have a type matching the Role Types at the end of the Relationship Type's definition.[1] Similarly, axiom (4) ensures that each Role which has a type at the end of a Relationship Type *rst* must be present in that relationship's extension, i.e. it must be in one of the tuples in the set returned by *link*(*rst*). In addition to that axiom (5) applies the cardinality constraints defined on the schema level to the instance level. In detail, this axiom ensures that ε is only allowed in a tuple if the corresponding cardinality has a lower bound of zero, i.e. $0..1$ or $0..*$, and that each role is related to at most one other role if the corresponding cardinality has an upper bound of one, i.e. $0..1$ or $1..1$. These rules are applied to both ends of the relationship with respect to the specific cardinalities. Consequently, this axiom ensures that each relationship respects the cardinalities defined on the schema level. Last but not least, axiom (6) ensures that whenever a role is linked to the empty counter role ε it is not linked to another role within the same relationship type. In sum, this definition permits the consistency of each instance i of a schema s.

Notably, ε represents an empty counter role, which can be replaced by a counter role later on. This is necessary, because each role participating in a relationship, i.e., its Role Type is at one end of a Relationship Type, must be in the extended set of that relationship. However, this leads to problems for relationships with at least one lower bound of zero indicating that one side of the relationship is not necessarily linked to the other side. To overcome this issue, we introduce ε as an empty placeholder for the missing counter roles for such relationships. This may result in Roles that are not related to any other Role except an empty counter Role, which enables additional flexibility in our model. For instance, the *Customer* Role Type can be related to a *CheckingsAccount* Role Type and a *Consultant* Role Type. For a new *Customer* instance neither a responsible *Consultant* nor a valid *CheckingAccount* may exist, because the account validation process is not finished yet. Enforcing a strict counter Role in the sense of foundation avoids insertion of a new *Customer* Role until the validation process is finished. Despite of that, a Role related to an empty counter Role is still considered founded, because its existence is still dependent on its player. As a result, ε is crucial to model relationships with a lower bound of zero.

Example 3 *Let* $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ *be the schema defined in Example 1. An instance of that schema is then* $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$*, where the components are defined as follows:*

$$N \stackrel{def}{=} \{\mathit{Peter}, \mathit{Klaus}, \mathit{Google}, \mathit{Account}_1, \mathit{Account}_2\}$$
$$R \stackrel{def}{=} \{\mathit{Con}, \mathit{Cu}_1, \mathit{Cu}_2, \mathit{CA}_1, \mathit{CA}_2, S, T\}$$

[1] Please note that $R^{\varepsilon}_{rt} = R_{rt} \cup \{\varepsilon\}$ enables the representation of roles to be related to ε.

$$
\begin{aligned}
\mathit{type} \stackrel{\mathit{def}}{=} \{&(\mathit{Peter} \to \mathit{Person}), (\mathit{Klaus} \to \mathit{Person}),\\
&(\mathit{Google} \to \mathit{Company}), (\mathit{Account}_1 \to \mathit{Account}),\\
&(\mathit{Account}_2 \to \mathit{Account}), (\mathit{Con} \to \mathit{Consultant}),\\
&(\mathit{Cu}_1 \to \mathit{Customer}), (\mathit{Cu}_2 \to \mathit{Customer}),\\
&(\mathit{CA}_1 \to \mathit{CheckingsAccount}), (\mathit{CA}_2 \to \mathit{CheckingsAccount}),\\
&(S \to \mathit{Source}), (T \to \mathit{Target})\}\\
\mathit{plays} \stackrel{\mathit{def}}{=} \{&(\mathit{Peter}, \mathit{Con}), (\mathit{Klaus}, \mathit{Cu}_1), (\mathit{Google}, \mathit{Cu}_2),\\
&(\mathit{Account}_1, \mathit{CA}_1), (\mathit{Account}_2, \mathit{CA}_2),\\
&(\mathit{Account}_2, S), (\mathit{Account}_1, T)\}\\
\mathit{links} \stackrel{\mathit{def}}{=} \{&(\mathit{owns} \to \{(\mathit{Cu}_1, \mathit{CA}_1), (\mathit{Cu}_2, \mathit{CA}_2)\}), (\mathit{transfer} \to \{(S, T)\}),\\
&(\mathit{advices} \to \{(\varepsilon, \mathit{Cu}_1), (\mathit{Con}, \mathit{Cu}_2)\})\}
\end{aligned}
$$

Like the schema $bank$, the instance i is simply created from Figure 2 by taking all the Naturals and Roles into account, map their types accordingly, link the Roles to their players, and assigning a tuple for each relationship in the figure. In addition to that, the instance must also contain a tuple for the Role Cu_1, because *Customer* participates in the *advices* Relationship Type. Consequently, this relationship contains a tuple relating Cu_1 to the empty counter role ε.

Definition 5 (Dynamic Tuple)**.** *Let $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema and $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$ an instance of s. Furthermore, $ddt = (nt, RT_{nt})$ is a Dynamic Data Type and $n \in N_{nt}$ a Natural of this type. A Dynamic Tuple d is then defined with respect to the set of Role Types $RT_n^d \subseteq RT_{nt}$ currently played by n:*

$$RT_n^d = \{rt \in RT \mid n \text{ plays } r \wedge \mathit{type}(r) = rt\}$$

Case 1 *$RT_n^d = \emptyset$: then $d = (n)$*
Case 2 *$RT_n^d = \{rt_1, \ldots, rt_m\}$:*
then the Dynamic Tuple is defined as $d = (n, R_1^d, \ldots, R_m^d)$ with

$$R_i^d = \{r \in R_{rt_i} \mid n \text{ plays } r\} \qquad \text{for all } i \in \{1, \ldots, m\} \text{ and } rt_i \in RT_n^d$$

Such a Dynamic Tuple d has exactly one Configuration $c^d = (nt, RT_n^d)$.

A Dynamic Tuple is built around a certain Natural and several Role sets, where each Role set holds Roles of a specific Role Type. Moreover, the Configuration of a Dynamic Tuple will change, if it starts playing a Role of a Role Type that has not been played or if it stops playing the only Role of the corresponding Role Type [JKVL14]. Finally, we can define whether two Dynamic Tuples are related by a certain relationship.

Definition 6 (Dynamic Relationships)**.** *Let $s = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ be a schema, $i = (N, R, \mathit{types}, \mathit{plays}, \mathit{links})$ an instance of s and $o, p \in N$ two naturals in i. The Dynamic Tuple $a = (o, R_1^a, \ldots, R_m^a)$ is then related to a Dynamic Tuple $b = (p, R_1^b, \ldots, R_n^b)$ with respect to a given Relationship Type $rst \in RST$, noted as $\mathbf{a}$ rst $\mathbf{b}$, iff there is an $i \in \{1, \ldots, m\}$ and a $j \in \{1, \ldots, n\}$ such that $\exists r_1 \in R_i^a,\ r_2 \in R_j^b\ .\ (r_1, r_2) \in \mathit{links}(rst)$.*

This definition allows to identify whether two Dynamic Tuples are related by a specific Relationship Type. Moreover, because each Natural can play Roles of the same type multiple times, Dynamic Tuples can be related to multiple Dynamic Tuples by the same Relationship Type. Consider, for instance, the instance i of the schema $bank$ containing the following Dynamic Tuples and Dynamic Relationships:

Example 4 *Let* $bank = (NT, RT, RST, \mathit{fills}, \mathit{rel})$ *be the schema (Example 1) and* $i = (N, R, \mathit{type}, \mathit{plays}, \mathit{links})$ *its instance (Example 3); then* i *contains the following Dynamic Tuples:*

$$d_{Peter} = (Peter, \{Con\}) \qquad d_{Klaus} = (Klaus, \{Cu_1\})$$
$$d_{Google} = (Google, \{Cu_2\})$$
$$d_{Account_1} = (Account_1, \{CA_1\}, \{T\}) \qquad d_{Account_2} = (Account_2, \{CA_2\}, \{S\})$$

Besides that, these Dynamic Tuples are related by the following Relationship Types:

$$d_{Klaus}\ owns\ d_{Account_1} \qquad d_{Google}\ owns\ d_{Account_2}$$
$$d_{Peter}\ advices\ d_{Google} \qquad d_{Account_2}\ transfer\ d_{Account_1}$$

In sum, this formal model captures not only Dynamic Data Types and Relationship Types but also Dynamic Tuples and Dynamic Relationships.

4 RSQL Extensions for Relationships

The query language is the interface users and applications use to interact with database management systems. Hence, extending the DBMS also requires the interfaces being extended to the new specifications. SQL has been designed to store data in and retrieve them from relations, thus, it directly operates on tables having no notion of relationships and roles. Usually, relationships in relational DBMS are represented as additional columns in tables or as a separate table. Depending on the cardinality one of these options is implemented. In the most general case, a N:M cardinality between the participants forms a separate table that is handled as a normal table having foreign key constraints as references to other tables. The DBMS simply cannot distinguish between tables storing entities and tables that store relationships. The other cases, 1:N and 1:1, lead to additional columns in tables that store entities, which means entities and relationships are mixed in that representation. Attributes of relationships can no longer be distinguished from entity attributes. It becomes worse, if cardinality changes, since the mapping of relationships depends on that cardinality. For instance, a 1:N cardinality is changed to N:M. That means, the additional columns introduced in one of the participant tables will be dropped and the relationship forms a separate table including the relationship attributes. All queries considering this relationship have to be rewritten, which makes database schema and queries fragile. Additionally, the resulting database schema significantly differs from the conceptual model, due to a lot of mappings to store the concepts and relationships relationally. That difference disables users to query for desired concepts without specific database schema knowledge.

⟨*select*⟩ ::= SELECT ⟨*projection-clause*⟩ FROM ⟨*from-clause*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*from-clause*⟩ ::= ⟨*config-expression*⟩ (, ⟨*config-expression*⟩)*
(, RELATING ⟨*relation-clause*⟩)*

⟨*config-expression*⟩ ::= (⟨*nt-name*⟩ ⟨*ntAbbreviation*⟩ | _)
(PLAYING ⟨*logical-derived-config-expression*⟩)?

⟨*logical-derived-config-expression*⟩ ::= ⟨*derived-config-exrpession*⟩
| NOT ⟨*logical-derived-config-expression*⟩
| (⟨*logical-derived-config-expression*⟩ ⟨*junctor*⟩
⟨*logical-derived-config-expression*⟩)

⟨*derived-config-exrpession*⟩ ::= ⟨*rt-name*⟩ ⟨*rtAbbreviation*⟩

⟨*relation-clause*⟩ ::= ⟨*rtAbbreviation*⟩ WITH ⟨*rtAbbreviation*⟩
USING ⟨*rst-name*⟩ ⟨*rstAbbreviation*⟩

Figure 3: Data Query Language Syntax

To enable users and applications to store and query for role-based data, we proposed RSQL as extension to SQL [JKVL14]. RSQL enables role specific integrity constraints on the query language level. It relies on Dynamic Data Types on the type level and Dynamic Tuples on the instance level, but without considering relationships between them. As shown before, traditional relational methods are infeasible for representing relationships properly in a relational DBMS. To overcome these limitations of traditional relational DBMS and to enable cardinality-independent first-class relationships, RSQL has to be extended. This extension comprises the creation, manipulation, and retrieval of relationships.

4.1 Data Query Language

The *Data Query Language* (DQL) is used to retrieve stored data from DBMS and in case of RSQL, qualified Dynamic Tuples are returned based on given Configurations. Figure 3 provides an overview on the select statement. In general, the select statement starts with 'SELECT' followed by the ⟨*projection-clause*⟩. This clause filters columns in the result set. After that, the ⟨*from-clause*⟩ describes Configurations based on ⟨*config-expression*⟩ and their relationships by the ⟨*relation-clause*⟩, followed by an optional ⟨*where-clause*⟩.

To handle complexity of Dynamic Data Types, RSQL provides ⟨*config-expression*⟩ as a sophisticated type description. A single ⟨*config-expression*⟩ describes a set of Configurations. Dynamic Tuples matching one of these Configurations will be added to the result. Specifying multiple ⟨*config-expression*⟩ in one select statement produces a Cartesian product. To filter the Cartesian product based on related Dynamic Tuples, the ⟨*relation-clause*⟩

```
SELECT p2.name, a.balance
FROM    Person p1 PLAYING Consultant co,
        Person p2 PLAYING Customer c,
        Account a PLAYING CheckingsAccount ca,
        RELATING co WITH c USING advices ad,
        RELATING c WITH ca USING owns o
WHERE ca.limit > 1000 AND p1.name="Peter";
```

Figure 4: Example Select Query

inside the ⟨*from-clause*⟩ is used. This is done with the prefix 'RELATING' followed by two Role Type abbreviations combined by a 'WITH'. To define which Relationship Type has to be used, its name and the corresponding alias after 'USING' have to be specified. This ⟨*relation-clause*⟩ defines, via which Role Types Dynamic Tuples have to be connected with each other. Additionally, the Relationship Type and its abbreviation have to be specified to enable projection operations on its attributes. This specification is also necessary, because Role Types can be connected multiple times. The Role Type abbreviations have to be defined in the ⟨*config-expression*⟩ of a Dynamic Data Type in advance. Of course, users can define multiple ⟨*relation-clause*⟩ to filter multiple connected Dynamic Tuples.

Implementing a ⟨*relation-clause*⟩ enables users to produce more robust queries than in SQL. Relationships become explicit to users, thus, they do not have to be mixed with other concepts. Attributes of relationships are addressed via the relationship itself. This enables cardinality free querying, since the user specifies a relationship and not its implementation within the DBMS. If the cardinality of a relationship changes conceptually, the DBMS schema will slightly change, only by altering the cardinality of the corresponding Relationship Type. But this means, no attributes have to be moved to other relations and additional tables are avoided. Thus, changing cardinality does not affect the query statement at all, but its results, so that applications have to handle the query result differently. Moreover, it is more intuitive for users writing queries with explicit relationships to distinguish between entities and their relationships.

Figure 4 illustrates a query for the *balance* of all *Customers* of *Peter* who has an *Account* limit that is higher than 1000. At first, three Configurations are specified, two are based on DDT *Person* and one is based on DDT *Account*. Afterwards, these Configurations are related by Relationship Types *advices* and *owns* within the *from-clause*. This filter aims at selecting only related Dynamic Tuples. Lastly, the result is filtered by the the *checkingsAccount* limit higher than 1000 and *Consultant's* name *Peter*.

4.2 Data Manipulation

The *Data Manipulation Language* (DML) in RSQL creates, deletes, and indirectly modifies Dynamic Tuples. Typically, Roles and Naturals are addressed by this language part and the DBMS automatically creates Dynamic Tuples. Adding Relationships between

⟨*insert-nt*⟩ ::= INSERT INTO ⟨*nt-name*⟩
(⟨*attribute-name*⟩ (, ⟨*attribute-name*⟩)*)
VALUES (⟨*value-expression*⟩ (, ⟨*value-expression*⟩)*)

⟨*insert-rt*⟩ ::= INSERT INTO ⟨*rt-name*⟩
(⟨*attribute-name*⟩ (, ⟨*attribute-name*⟩)*)
VALUES (⟨*value-expression*⟩ (, ⟨*value-expression*⟩)*)
OF ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*insert-rst*⟩ ::= INSERT INTO ⟨*rst-name*⟩ ((⟨*attribute-name*⟩ (, ⟨*attribute-name*⟩)*)
VALUES (⟨*value-expression*⟩ (, ⟨*value-expression*⟩)*))?
INSTANCES ⟨*rtAbbreviation*⟩ , ⟨*rtAbbreviation*⟩
FROM ⟨*config-expression*⟩ , ⟨*config-expression*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*update-rst*⟩ ::= UPDATE ⟨*rst-name*⟩
SET ⟨*assignment-expression*⟩ (WHERE ⟨*where-clause*⟩)?

⟨*delete-rst*⟩ ::= DELETE FROM ⟨*rst-name*⟩ (WHERE ⟨*where-clause*⟩)?

Figure 5: Data Manipulation Language Syntax

Roles to the system implies large-scale changes of how Dynamic Tuples are handled. The DBMS, as single point of truth in a distributed software system, has to ensure consistency and in case of RSQL it has to ensure role-specific integrity conditions. This also includes ensuring the cardinality of relationships. For instance, Roles of Role Types that have to be part of a relationship cannot exist on their own. The example illustrated in Figure 1 shows such a Relationship Type. There, a *Source* and *Target* Role Type are connected by a *transfer* Relationship Type having 1..1 and 1..1 cardinality. This means, for each *Source* there has to be a *Target* and neither of them can exist without being in the *transfer* Relationship. That small example requires two Roles to be created at the same time and relating them. To ensure consistency with respect to the model, the insert operations must be embedded in a transaction and at the end of each transaction the validity is checked.

The DML comprises insert, update, and delete statements for Naturals, Roles and Relationships. In Figure 5 the syntax of relationship focused statements is presented. A Natural can be added to the system by using the ⟨*insert-nt*⟩ statement. It is comparable to inserting a tuple into a relational table using SQL. The ⟨*insert-rt*⟩ statement extends a Dynamic Tuple by a certain Role. It also comprises an attribute and value assignment as well as a configuration description to determine the Dynamic Tuple that has to be extended. In addition to these statements, the new ⟨*insert-rst*⟩ statement has been introduced which relates two existing Roles of Dynamic Tuples to each other. The process of relating Dynamic Tuples is the following: Firstly specify which Role Type the Roles belong to, secondly specify the Dynamic Tuples in which the corresponding Roles are present, and thirdly filter the particular Roles. The statement also starts with 'INSERT INTO' followed by the Relationship Type's name. Afterwards, users can specify attributes and values for those

```
BEGIN TRANSACTION;
INSERT INTO Source (id, tan) VALUES (1, 0321) OF
        Account a WHERE a.iban = "0815";
INSERT INTO Target (id) VALUES (2) OF
        Account a WHERE a.iban = "4711";
INSERT INTO transfer (creation, execution, amount)
        VALUES ("11.09.2014", "12.09.2014", 500)
        INSTANCES s, t
        FROM    Account a1 PLAYING Source s,
                Account a2 PLAYING Target t
        WHERE s.id = 1 AND t.id = 2;
COMMIT;
```

Figure 6: Data Manipulation Language Examples within a Transaction

in an optional clause. The attribute and value assignment is optional, since Relationship Types without attributes may exist. Then, the first process step follows by stating the corresponding Role Type with their abbreviations in the 'INSTANCE OF' clause by. Afterwards, the 'FROM' clause follows having exactly two ⟨*config-expression*⟩ to specify the second process step. There, each Role Type stated in the first step has to be present in one ⟨*config-expression*⟩. So far, users defined via which Role Types the resulting Dynamic Tuples have to be connected. In case Roles of the same Role Type are present multiple times in the same Dynamic Tuple, the 'WHERE' clause has to filter exactly one of those Roles to determine the unique Role that will be related. In general, inserting a Relationship is independent of the Roles that might have been inserted in the same transaction, it connects the Roles that have been described in the 'FROM' and 'WHERE' clauses.

The ⟨*update-rst*⟩ statement updates values in Relationships. It is similar to updating Naturals or Roles. Users have to specify 'UPDATE' and the corresponding Relationship Type name at the statement's very beginning. The system will figure out for itself which type has to be updated by the user provided name, since names have to be unique over all types. To delete Relationships and detach the corresponding Roles, the ⟨*delete-rst*⟩ statement has to be used. The syntax of this statement is similar to traditional delete statements, however, the Relationship Type name has to be specified. If a Relationship is deleted, Roles participating in this Relationship may also be deleted in case they are constrained with a lower bound cardinality of 1 and the last participating Role on one relationship side. This can cause knock-on effects for a series of other Roles and Relationships and the user must be aware of this. Roles participating in this particular Relationship as optional with a lower bound cardinality of 0, are not affected. The other way around, a deletion of a Role may also cause the deletion of a Relationship.

The statements in Figure 6 show an example for inserting two Roles and relating them within a transaction. At first, we create a new transaction, because the following insertions will lead to inconsistent states during the transaction. Afterwards, two Roles are inserted into the database. After each insertion, the stored data is invalid with respect to the schema, that requires that Source and Target have to be connected to exactly one counter Role. To

⟨*create-nt*⟩ ::= `CREATE NATURALTYPE` ⟨*nt-name*⟩
`(` ⟨*attribute-definition*⟩ `(` `,` ⟨*attribute-definition*⟩ `)*` `)`

⟨*create-rt*⟩ ::= `CREATE ROLETYPE` ⟨*rt-name*⟩
`(` ⟨*attribute-definition*⟩ `(` `,` ⟨*attribute-definition*⟩ `)*` `)`
`PLAYED BY` ⟨*nt-name*⟩ `(` `,` ⟨*nt-name*⟩ `)*`

⟨*create-rst*⟩ ::= `CREATE RELATIONSHIPTYPE` ⟨*rst-name*⟩
`(` `(` ⟨*attribute-definition*⟩ `(` `,` ⟨*attribute-definition*⟩ `)*` `)` `)?`
`CONSISTING OF (` ⟨*relation-participation*⟩ `)` `AND (` ⟨*relation-participation*⟩ `)`

⟨*relation-participation*⟩ ::= ⟨*rt-name*⟩ `BEING (` `0` | `1` `) . . (` `1` | `*` `)`

⟨*drop-rst*⟩ ::= `DROP RELATIONSHIPTYPE` ⟨*rst-name*⟩

Figure 7: Data Definition Language Syntax

restore the validity we insert a new Relationship of the type *transfer*. In the first process step we state the desired Role Types. Afterwards, we define two distinct Dynamic Tuples in the FROM clause, one playing a *Source* Role and one playing a *Target* Role. Usually, this will not be the first transfer on the corresponding accounts, which implies that both Dynamic Tuples of the *Accounts* probably have multiple *Source* and *Target* Roles already. To uniquely identify the desired Roles, we filter the Dynamic Tuples in the '`WHERE`' clause by the *Source id* and *Target id* attribute. Finally, the transaction will be checked and, if valid, committed.

4.3 Data Definition Language

The *Data Definition Language* (DDL) specifies the schema of a database. There, all type information will be provided to the DBMS. Dynamic Data Types are created, altered and dropped indirectly. Creating a new Natural Type creates a new Dynamic Data Type and creating a new Role Type extends existing ones. Role Types, as parts of Dynamic Data Types, can be related by Relationship Types. Also cardinality of Relationship Types is defined using DDL statements.

A Natural Type is created by a ⟨*create-nt*⟩ statement. This statement starts with '`CREATE NATURALTYPE`' followed by a distinct name and a set of attribute definitions. Note, all types form a common name space, such that each defined type has a distinct name. An existing DDT is extended by Role Types using the ⟨*create-rt*⟩ statement. After '`CREATE ROLETYPE`' and a unique name, attributes have to be specified, and at least one Natural Type has to be given after '`PLAYED BY`' clause. This clause avoids isolated Role Types. Alter and drop statements also exist for those types, but are out of scope here.

To relate Dynamic Data Types with each other, Relationship Types have to be used. This

```
CREATE RELATIONSHIPTYPE advices CONSISTING OF
        (Consultant BEING 0..*) AND (Customer BEING 1..*);
CREATE RELATIONSHIPTYPE transfers
        (creation Timestamp, execution Timestamp, amount Money)
        CONSISTING OF (Source BEING 1..1)
        AND (Target BEING 1..1);
```

Figure 8: Data Definition Language Examples

enables the DBMS to handle relationships explicitly and provides information about relationship attributes and cardinality. This information will be used by the DBMS to check the validity of DML operations. A Relationship Type is created using the ⟨*create-rst*⟩ statement. It starts with 'CREATE RELATIONSHIPTYPE' and a unique name. After that, it is optional to specify attributes for this Relationship Type. This is optional, since Relationship Types without attributes may exist. To relate Role Types two ⟨*relation-participation*⟩ clauses have to be combined with a prefix 'CONSISTING OF'. The first ⟨*relation-participation*⟩ denotes the left part of the Relationship Type and the second one the right part. A ⟨*relation-participation*⟩ consists of a Role Type name and its cardinality. The Role Type name specifies the corresponding Role Type that will become part of this relationship. The cardinality is constructed using 'BEING' followed by the lower bound. There, users can choose between "0" and "1". Afterwards, the upper bound is defined while having two options: "1" or "*". Relationship Types can be altered and dropped as well, but for reasons of brevity we focused on creation only. RSQL also provides a statement to alter Relationship Type attributes, the referenced Role Types, and cardinality, but here we focused on the creation of Relationships.

Figure 8 illustrates syntax and semantics of the DDL. At first, the type level is created assuming that the Dynamic Data Type *Person* and DDT *Account* already exist. The first statement establishes a Relationship Type *advices* between *Consultant* and *Customer* without attributes. Moreover, the *Consultant*'s cardinality is 0 to $*$ and the *Customer*'s is 1 to $*$ denoting that each *Consultant* needs at least one *Customer*. A *Consultant* will be forced to be related to a *Customer* and has to participate in such a relationship. In contrast, the *Customer* can exist without this relationship, but can also have more than one Consultant. The second statement creates a new Relationship Type *transfer* between *Source* and *Target* with three attributes as well as 1 to 1 and 1 to 1 cardinality, respectively.

5 Comparison

The role concept has been proposed by Bachman in 1977 [BD77]. Over the past decades the idea of separating different concerns of data objects has been adapted to many modeling languages, especially in conceptual modeling. Henceforth, we first classify our approach and discuss other related query languages, afterwards.

5.1 Classification

To classify our role-based data model, we employ a previously defined classification scheme developed to evaluate role-based modeling and programming languages [Ste99] and its extension [KLG+14]. Consequently, we can classify RSQL's formal model in accordance with the features of roles: *(1) Roles have properties and behaviors.* Yes, roles have properties. *(2) Roles depend on relationships.* Yes, roles can be defined to depend on relationships. *(3) Objects may play different roles simultaneously.* Yes, by definition. *(4) Objects may play the same role (type) several times.* Yes. *(5) Objects may acquire and abandon roles dynamically.* Yes. *(6) The sequence of role acquisition and removal may be restricted.* Partially, because cardinality constraints can be used to restrict the order of role creation. *(7) Unrelated objects can play the same role.* Yes. *(8) Roles can play roles.* No. Similar to *Lodwick* we disregard roles playing roles. *(9) Roles can be transferred between objects.* Yes, this can be done by altering the player of a role. *(10) The state of an object can be role-specific.* Yes, because a Dynamic Tuple combines the state of the natural and its roles. *(11) Features of an object can be role-specific.* Yes, because an object is defined as a Dynamic Data Type. *(12) Roles restrict access.* This feature is not applicable, because we currently assume that each user has full access to all defined Dynamic Data Types. *(13) Different roles may share structure and behavior.* No, currently, our formal model does not support inheritance between roles. *(14) An object and its roles share identity.* Yes, because they are encapsulated in a Dynamic Tuple inheriting its identity from the object. *(15) An object and its roles have different identities.* Yes, they have different identities in the formal model. *(16) Relationships between roles can be constrained.* Yes, by supporting the definition of cardinality constraints. *(17) There may be constraints between relationships.* No, currently the definition of inter relationship constraints is not allowed. *(18) Roles can be grouped and constrained together.* No. *(19) Roles depend on compartments.* No, there is no notion of Context or Compartment in our model. In consequence, all other features, i.e., Feature 20 to Feature 26, can not be fulfilled at all. As a result, this apporach focuses on the relational nature of roles, but allows for shared as well as own identities for roles by introducing Dynamic Data Types.

5.2 Related Work

Henceforth, we recollect the various related approaches with respect to the underlying data model and role-based query language.

Considering the former, the definition presented here has similarities to *LODWICK*'s definition in [Ste99]. Both are founded on Naturals and Roles on the type level, but on the instance level roles do not exist in *LODWICK*. Furthermore, *LODWICK*'s formal model is able to express arbitrary n-ary relationships, RSQL's data model can only express binary ones. Additionally, Roles and Naturals in RSQL have different identities, but they are combined to Dynamic Data Types, which can be identified solely by a specific Natural. However, Dynamic Tuples in RSQL are identified by value based identification rather than unique object ids, i.e., they inherit their identity directly from the contained Natural.

In direct comparison to *LODWICK*, RSQL is superior because it can resolve the identity issue of roles and includes the definition of cardinality constraints.

The related query languages can be grouped along three different classes: role-based query languages, conceptual query languages, and regular query languages. In the first class, the *Information Networking Model* (INM) uses roles in relationships to describe that objects play roles in a certain relationship to other objects [LH09a, LH09b]. Like RSQL and the underlying formalism, it also supports dynamic and many-faceted object types. Furthermore, INM provides features to model context-dependent information and thus, it introduces contexts to group roles. Relationships also exist in INM, first as normal relationships or role relationships. Via role relationships context information is modeled and normal relationships are the same as in traditional modeling languages. Nevertheless, relationships cannot be constrained by cardinality. Moreover, there exists a query language for INM, called *Information Query Language* (IQL) [HFL10]. The IQL utilizes tree expressions, like *XPath*, to hierarchically describe the desired data. Moreover, they describe a DBMS that persists INM-based data in a key-value-store (Berkeley DB). In contrast, RSQL aims at a relational DBMS to take advantage of the richer role semantics to (i) store role-based data more efficiently and (ii) optimize queries. Key-value-stores cannot take advantage of in-DBMS optimizations, because no information about the stored data is known to the DBMS. Optimizations have to be performed outside of the DBMS, which causes overhead in transferring and processing the data.

Conceptual query languages abstract implemented database models to conceptual models. Hence, users query based on conceptual levels without any knowledge about the implemented data representation. This approach has become popular in the 1980's and 1990's. At that time, a lot of entity-relationship based conceptual query languages for relational DBMS have been proposed, since data were modeled using ER and stored relationally. Examples of query languages based on ER or extended ER are *SQL/EER* [HE92], the query language presented in [LT94], or *CABLE* [Sho79]. Nevertheless, these query languages are usually mapped to SQL, thus they only abstract for users and relational DBMS remain the same. Moreover, they have no notion of roles but of relationships and RSQL comprises both notions. However, query languages exist that have a notion of roles, for instance *ConQuer* [BH96] that is designed for *Object Role Modeling* (ORM) [Hal98]. In contrast to RSQL, *ConQuer* queries are also mapped to SQL.

In comparison to general query languages, RSQL enables users and applications to query information based on a conceptual model, which relates it to conceptual query languages. Nevertheless, RSQL does not abstract database objects to conceptual entities. RSQL is a regular query language that directly describes database objects that are handled in a DBMS without abstraction. This also requires introducing new database objects in a DBMS and creating new first-class citizen. The only approach that has introduced roles as first-class citizen in a DBMS is *DOOR* [WCL97]. DOOR also has the notion of rigid and non-founded types (Natural Types in RSQL) and founded, non-rigid types (Role Types) on the type level as well as on the instance level. However, DOOR does not support relationships between roles.

6 Conclusions

To lower the mismatch between role-based conceptual models and their DBMS implementation, we extended our data model and query language RSQL by introducing first-class relationships. Hence, role and relationship semantics are preserved in the DBMS and not hidden in mapping engines. This improves the interoperability in highly distributed information systems that run role-based software and allows DBMS to guarantee role specific as well as relationship specific consistency constraints. In detail, we introduced Relationship Types on the type level and Relationships on the instance level in our data model. Additionally, cardinality constraints are directly bound to this newly introduced types. Consequently, RSQL has been adapted to the new data model by introducing special statements to create Relationship Types and insert Relationships, respectively. By making relationships explicit, they do not have to be mapped to tables and queries can be created independently of the relationship implementation. This has the following two effects. Firstly, cardinality changes no longer entail query reconstruction, which produces more robust queries. Secondly, users can create queries without having knowledge about the relational mapping and normalization of relationships. Additionally, the DBMS gains knowledge on the stored data objects that can be used to optimize query processing and enforce relationship consistency constraints. In summary, RSQL enables users and applications to directly represent and query for role-based data objects including their relationships and, at the same time, the implementing DBMS can ensure role-based consistency constraints.

We have presented a formal data model definition and a query language that captures that definition. To constrain Dynamic Data Types and the set of Configuration they describe, we will introduce constraints between Role Type (e.g. prohibition of two Role Types). Additionally, we aim at cardinality constraints on the fills-relation to represent real one-to-one relationships in our system. As a next step, we will implement both the data model and RSQL within an extended relational DBMS. Hence, we will introduce new database objects to teach the DBMS the notion of Dynamic Data Types and Relationship Types. To process Dynamic Tuples, we are going to implement specialized database operators that can be used for query processing as well as for query optimization.

Acknowledgement

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

References

[BD77] Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *Proceedings of the third International Conference on Very Large Data Bases*, pages 464–476. VLDB Endowment, 1977.

[BH96] A.C. Bloesch and T.A. Halpin. ConQuer: A Conceptual Query Language. In *Proceedings of the International Conference on Conceptual Modeling – ER '96*, volume 1157 of *Lecture Notes in Computer Science*, pages 121–133. Springer Berlin Heidelberg, 1996.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[Gua92] Nicola Guarino. Concepts, Attributes and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases. *Data & Knowledge Engineering*, 8(3):249 – 261, 1992.

[Hal98] Terry Halpin. ORM/NIAM Object-Role Modeling. In *Handbook on Architectures of Information Systems*, International Handbooks on Information Systems, pages 81–101. Springer Berlin Heidelberg, 1998.

[HE92] Uwe Hohenstein and Gregor Engels. SQL/EER — Syntax and Semantics of an Entity-relationship-based Query Language. *Information Systems*, 17(3):209–242, May 1992.

[HFL10] Jie Hu, Qingchuan Fu, and Mengchi Liu. Query Processing in INM Database System. In Lei Chen, Changjie Tang, Jun Yang, and Yunjun Gao, editors, *Web-Age Information Management*, volume 6184 of *Lecture Notes in Computer Science*, pages 525–536. Springer Berlin Heidelberg, 2010.

[JKVL14] Tobias Jäkel, Thomas Kühn, Hannes Voigt, and Wolfgang Lehner. RSQL - A Query Language for Dynamic Data Types. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 185–194, New York, USA, 2014. ACM.

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-based Modeling and Programming Languages. In *7th International Conference on Software Language Engineering (SLE)*. Springer, 2014.

[LH09a] Mengchi Liu and Jie Hu. Information Networking Model. In *International Conference on Conceptual Modeling – ER 2009*, volume 5829 of *Lecture Notes in Computer Science*, pages 131–144. Springer Berlin Heidelberg, 2009.

[LH09b] Mengchi Liu and Jie Hu. Modeling Complex Relationships. In *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 719–726. Springer Berlin Heidelberg, 2009.

[LT94] Michael Lawley and Rodney Topor. A Query Language for EER Schemas. In *Proceedings of the 5th Australian Database Conference*. Global Publication Service, 1994.

[RJB10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (Paperback)*. The Addison-Wesley object technology series. ADDISON WESLEY Publishing Company Incorporated, 2010.

[Sho79] Arie Shoshani. CABLE: A Language Based on the Entity-Relationship Model. Technical report, Lawrence Berkeley Laboratory, 1979.

[Ste99] Friedrich Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83 – 106, 1999.

[WCL97] R.K. Wong, H.L. Chau, and F.H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 402–411, Apr 1997.

SemRep: A Repository for Semantic Mapping

Patrick Arnold, Erhard Rahm

University of Leipzig
arnold@informatik.uni-leipzig.de
rahm@informatik.uni-leipzig.de

Abstract: In schema and ontology matching, background knowledge such as dictionaries and thesauri can considerably improve the mapping quality. Such knowledge resources are especially valuable to determine the semantic relation type (e.g., equal, is-a or part-of) that holds between related concepts. Previous match tools mostly use WordNet as their primary resource for background knowledge, although WordNet provides only a limited coverage and currentness. We present the design and use of a new comprehensive repository called *SemRep* that combines concepts and semantic relationships from different resources. It integrates both manually developed resources (including WordNet) and semi-automatically extracted relations from Wikipedia. To determine the semantic relationship between two concepts of interest, SemRep also considers indirect relationships of possibly different types. An initial evaluation shows the general effectiveness and efficiency of using SemRep for ontology matching.

1 Introduction

Background knowledge plays an important role for semantic data interoperability, in particular to automatically determine schema and ontology mappings. Many generic match strategies have been developed in the past to determine related concepts between two schemas resp. ontologies, most of which utilize the lexical similarity of element names, the structural similarity of elements or the similarity of associated instance data [BBR11]. However, such strategies often fail in real-world scenarios, e.g., for synonymous concepts with no or low lexicographic similarity (e.g. *car/automobile*) or in the presence of homonyms like *mouse (computer) / mouse (animal)*. A main solution to this issue is the use of background knowledge such as thesauri, making it easy to look-up and use synonymous concept names. Thesauri can also help to determine more complex relationships between concepts of different ontologies, such as is-a and part-of relations.

Unfortunately, there is only a small number of background knowledge resources suitable for matching. There are some elaborated background knowledge ontologies for specific domains, like UMLS for the medical domain, but for general matching scenarios only few resources exist so far. WordNet is among the most popular ones, but is still rather limited in size and currentness. For example, we discovered that WordNet does not contain specific concepts we wanted to match, e.g., furniture concepts such as *teapoy* (a little table with three legs), *cassone* (a historical chest) or *basinette* (synonym for cradle). While these con-

cepts are represented in Wikipedia, they are also missing in Wikipedia-based knowledge resources such as DBpedia or Yago which focus on entities (persons, geographic places, movies etc.), rather than concepts (see discussion in Related Work). As a result we are missing a comprehensive knowledge resource about concepts and their relations, e.g., to help determine mappings between schemas and ontologies.

For this reason, we are building up a comprehensive semantic repository called *SemRep* combining background knowledge from diverse sources. The focus is on collecting concepts and their semantic relations (equal, is-a, part-of and their inverses) in order to support automatic schema and ontology matching, albeit further use cases such as term disambiguation or text mining may also be served.

Table 1 provides details about the four resources that are currently integrated in SemRep, including the number of concepts and relations we gain from those resources. We combine knowledge from three curated resources including WordNet. By far most concepts and relations are derived from an automatic extraction from Wikipedia, based on our methods described in [AR14b]. SemRep is thus many times more comprehensive than WordNet so that it promises a much better coverage for matching tasks. This is also because SemRep can derive the semantic relation type for indirectly related concepts by evaluating many possible relation paths between concepts, e.g., *automobile equal car is-a vehicle*. Evaluating such paths poses several challenges for efficiency and determining the correct relation type, especially for paths with different relation types.

Resource	**Lang.**	**Creation**	**#Concepts**	**#Relations**	**File size**
WordNet	English	Manually	116,326	1,694,505	45 MB
Wikipedia	English	Automatically	1,051,170	2,843,428	149 MB
UMLS	English	Manually	109,599	281,972	19 MB
OpenThesaurus	German	Manually	58,473	914,864	25 MB

Table 1: Resources for the imported relations used in the repository.

We make the following contributions:

- We present the design and implementation of SemRep, an extensible repository combining mapping-related background knowledge from multiple resources of different domains.
- We discuss how relations between two concepts can be directly and indirectly resolved using the repository, and what obstacles may arise.
- We evaluate the usefulness of SemRep for different ontology matching tasks. The evaluation is performed with our semantic ontology matching tool STROMA that determines correspondences with their semantic relation type (equal, is-a, inverse is-a, has-a, part-of or related) [AR14a]. We will show in our evaluation how the use of SemRep can improve the mapping quality compared to the sole use of WordNet.

Our study is organized as follows: Section 2 discusses related work. Section 3 gives an overview on our repository architecture while section 4 discusses the execution of queries

and the determination of relation types. We will evaluate our approach in section 5 and conclude in section 6.

2 Related Work

Background knowledge is widely used in schema and ontology matching and helps considerably to improve the quality of the resulting mappings [AKtKvH06]. Background knowledge includes dictionaries, thesauri, and domain-specific taxonomies, e.g., UMLS for the medical domain, as well as existing schemas and mappings (corpus-based matching) [MBDH05, GHKR11]. These resources are especially valuable for determining semantic mappings consisting of correspondences of different relation types (is-a, part-of, etc.). Some tools like COMA and GOMMA also exploit previously generated match results as background knowledge to derive new correspondences, which is referred to as mapping reuse [DR02, GHKR11] For example, the composition of a given mapping between schemas S_1, S_2 with an existing mapping between schemas S_2, S_3 leads to a new mapping between S_1 and S_3 if one assumes transitivity of correspondences.

Background knowledge resources are developed either manually or automatically. In the first category, resources are created and curated either by individual experts or collaboratively by a community of volunteers (e.g., OpenThesaurus). In the linguistic domain, WordNet is certainly the most popular resource for the English language [GM98]. It was manually developed by linguistic experts and thus has a high quality. However, WordNet is relatively limited w.r.t. size and currentness. The latest available version is from 2006 and misses many modern terms like *netbook* or *smart phone*.

Resources with automatically generated background knowledge promise a much better coverage and currentness, but typically face significantly more quality issues than manually controlled resources. We devised an automatic approach to extract concepts and semantic relations between them from Wikipedia [AR14b]. This is achieved by analyzing the first (definitional) sentence of each Wikipedia article and looking for specific semantic patterns like *A is a specific form of B*. The resulting relations have been included in the SemRep repository (Table 1). A similar approach was used in [WLWZ12] to extract is-a relations from more than 1.7 billion websites resulting in a large taxonomy of 2.7 million concepts. So far, such automatically generated background knowledge is not yet exploited for schema and ontology matching. Existing tools mostly use WordNet as their primary resource, in particular for semantic matching (e.g. in S-Match [GAP12] or TaxoMap [RS07]).

Several recent approaches focus on the integration of thesauri from different languages. For example, EuroWordNet combines WordNet with thesauri from eight European languages. UBY-LMF is a relatively new framework to represent linguistic relations and dictionary data for many languages, which may differ greatly w.r.t. flexion, grammatical cases, morphology etc. [EKGH$^+$12]. BabelNet also combines linguistic relations for many languages by aligning Wikipedia pages to WordNet concepts [NP10]. In contrast to these approaches, we do not focus on integrating knowledge for different languages but

rather on comprehensively integrating semantic relations between concepts from several sources.

SemRep also differs from knowledge bases providing information about entities (like persons, countries, movies, books, music albums, buildings etc.) rather than concepts. Entities are commonly less related by semantic relations, but primarily by specific relations like *wasBornIn*, *livesIn*, *wasFoundedBy*, *isDirectorOf* etc. Popular resources of this kind include DBpedia [ABK+07], Freebase [BEP+08] and the more domain-specific Geonames. Yago is a special resource, as it combines DBpedia with WordNet, so a knowledge base with a linguistic resource [SKW07]. Still, it provides little advantage over WordNet for supporting schema and ontology matching.

Sabou et al. describe a different way of using background knowledge. Instead of exploiting local resources, they resolve correspondences dynamically by using the ontology search engine Swoogle in order to find relevant background knowledge ontologies for a mapping scenario at hand [SdM06]. They are facing similar problems as we do, like how to determine the relation type in paths larger than 1, or how to deal with contradicting results. Such web-based approaches are generally slower than approaches using local resources, however, the authors did not evaluate the performance or execution times of their approach.

3 Repository Overview

Our goal is to provide with SemRep a comprehensive repository of semantic concept relations in order to support ontology and schema matching. We focus on the general relations listed in Table 2. To achieve a broad coverage we aim at an extensible design making it easy to include the knowledge from several resources. As already mentioned, we have so far covered knowledge from the four resources listed in Table 1 so that SemRep provides semantic relations for concept names from two languages (English and German).

Relation type	Example	Linguistic relation
equal	river, stream	Synonyms
is-a	car, vehicle	Hyponym – Hypernym
inverse is-a	computer, laptop	Hypernym – Hyponym
has-a	body, leg	Holonym – Meronym
part-of	roof, building	Meronym – Holonym

Table 2: Supported relation types by SemRep.

SemRep must be able to quickly determine for two concepts a and b the semantic relation between these concepts if they are recorded in the repository. This is a simple lookup query if a and b are interrelated by a single relation that was provided by one of the input resources. However, the real usefulness of SemRep comes from the possibility to combine several relations, possibly from different input resources, to indirectly derive semantic relations. Due to the different kinds of relations this is a non-trivial task and we describe

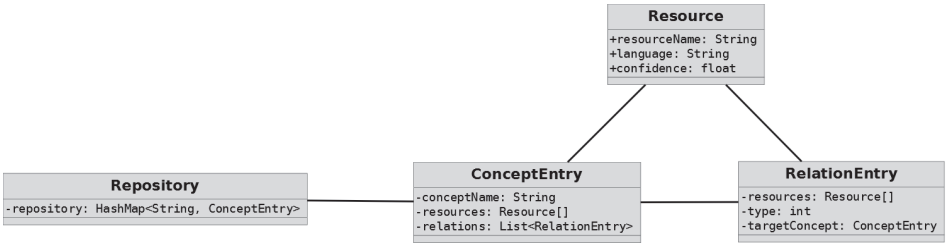


Figure 1: Repository infrastructure in UML notation.

in the next section how we address it. There are also high efficiency requirements, since all queries should be answered very quickly to be usable for matching large ontologies with possibly ten thousands of concepts.

In the rest of this section, we briefly discuss the current implementation approach and how we import semantic relations into the repository.

3.1 Implementation aspects

SemRep represents concepts and their semantic relations in a large directed graph structure where nodes refer to concept names and edges to semantic relations. In an initial implementation we tried to use a database system to maintain this graph, especially a relational DBMS (mySQL) as well as a graph DBMS (Neo4j). Unfortunately, we could not achieve acceptable query execution times in both cases. For example, it took up to 30 seconds with the graph DBMS to determine an indirect correspondence over three steps.

We therefore decided to develop a tailored repository implementation utilizing a Java-based hash map data structure to store the concept nodes. The implementation adopts the simple UML model shown in Fig. 1. Each concept entry has a name, a list of resources where it appeared and a list of relation entries. A relation entry has a specific type, a list of resources where it appeared, as well as a target concept referring to an existing concept in the hash map. As an example, we may have the concept car with two relations to $automobile$ and $vehicle$. Let us assume that the first relation was provided by WordNet, the second by Wikipedia. The concept is thus represented as (car, <WordNet, Wikipedia>, $< r_1, r_2 >$) with the two relations being r_1 = (equal, <WordNet>, automobile) and r_2 = (is-a, <Wikipedia>, vehicle).

Since we can keep the hash map in main memory, this simple approach achieves very fast query execution times despite the large number of concepts and relations.

3.2 Data Import

SemRep makes it easy to include semantic relations from about any source by importing the relations in a simple triple format: (source concept, target concept, relation type). Listing 1 shows a sample extract from an import file; relation types are encoded by a single digit to facilitate the parsing (0 = equal, 1 = is-a, 4 = part-of). To include a new resource to the repository, its semantic relations have to be written into a correctly formatted text file. The file importer can then import these relations and add them to the repository. For is-a and part-of relations, we also add the inverse relations. The simple file format and repository structure make it very fast to launch and populate SemRep. Loading the four resources into the repository takes only about 75 seconds, i.e., we import about 180,000 relations per second. 9 GB of RAM are needed for the graph structure.

Listing 1: Excerpt from an import file

```
car :: vehicle :: 1
car :: automobile :: 0
mountain bike :: bike :: 1
bike ring :: handlebars :: 4
```

There are several options to configure the data import for improved flexibility and extensibility. First it can be specified what should be imported, e.g., only English resources or only WordNet. For each resource, the language and a confidence value has to be specified, e.g., to apply higher confidence values for manually curated resources like WordNet than for automatically generated input such as our Wikipedia relations (see next section). There is also an optional import filter to block relations which may cause problems for query execution. We can therefore filter relations that are involved with general terms such as *object*, *unit*, *entity*, *thing* or *abstraction*. Relations like (furniture is-a object) are meaningless, and since everything may be an object, such relations could lead to false conclusions when querying a semantic relation. We therefore use a blacklist of concepts which we dot not wish to have in our repository and block respective relations. In the future, we plan additional tests to improve data quality, e.g., to check whether imported relations cause inconsistencies. To avoid an increased launch time, we can eliminate such problematic relations from the input file to avoid their import in future launches of the repository.

The number of English-language concepts in the repository after filtering is 1,219,233 and the number of relations is 4,553,688, which means an average number of 3.7 relations per concept. 29 % of all relations are of type equal, 42 % of type is-a or inverse is-a, and 28 % of type part-of or has-a.

4 Querying SemRep

Executing queries to determine the relation type holding between two terms a, b consists of the four steps shown in Fig. 2. The first and last step, indicated by dashed lines, is optional while the second and third step is always carried out. We start with some preprocessing on the terms in case that at least one of the two terms is not contained in the repository. Next, we retrieve a set of paths from a to b within a specified maximal path length. We then determine for each result path a path type and a score (step 3). If no valid path is found, we apply some post-processing, trying to still find some valid path. Finally, the path type of the highest-scored path is returned.

In the following, we describe the introduced steps.

4.1 Preprocessing

Apparently, a relation between concepts a, b cannot be found if the repository is not containing one of the terms as it was not provided by any input resource. This is often the case for very specific concepts which are most frequently compound words that are too obvious to be listed in any resource. For instance, the two words *kitchen* and *chair* can be expected to appear in any English dictionary or thesaurus. The rather simple compound *kitchen chair*, which is obviously a specific chair usually found in the kitchen, is not contained by most dictionaries (including WordNet, Wikipedia, Wiktionary, the Oxford Dictionary and the Free Dictionary), because of its simplicity and because of the huge amount of such compound words that could be created.

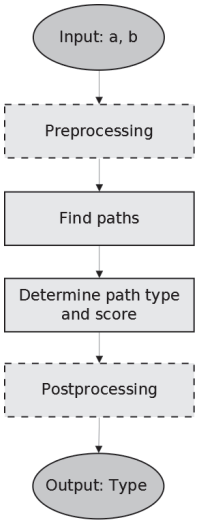


Figure 2: Querying Workflow.

In matching, however, such compound words are of critical importance, and treating compounds correctly can help to overcome the issue that terms are not contained by the repository. We have already shown in [AR13] that a compound consists of a *head* C_H, which is the right-hand word of the compound and specifies its basic meaning, as well as at least one so-called *modifier* C_M appearing left of the head. The compound is normally a specification of the compound head, thus a *kitchen chair* is a specific form of a *chair*. We proposed a technique called *gradual modifier removal* (GMR) to gradually remove the modifiers of the compound until one finally finds the word in the dictionary. If we want to determine the relation type between *kitchen chair* and *seat*, we would remove the modifier *kitchen* from *kitchen chair* and see that *chair* is contained by our repository. The repository suggests the relation *chair is-a seat*,

and since a kitchen chair is a specific chair, we can conclude the relation *kitchen chair is-a seat*. With this, we have determined a correct semantic relation type, even though the term *kitchen chair* is not contained in our repository.

Apart from that, we implemented two additional techniques for special compound correspondences. We noticed two important cases:

1. Two compounds can have the same modifier, e.g., *US vice president* and *US politician*.
2. Two compounds can have the same head, e.g., *apple cake* and *fruit cake*.

The first case is relatively easy. If the repository does not contain both input words, we remove the two identical modifiers of both compounds and end up with comparing *vice president* and *politician*, which are both in the repository. The second case is a little more complicated. We can remove the compound head of both concepts and focus our comparison on the two remaining modifiers if, and only if, they are synonyms or in a hypernym relation. For instance, an *apple* is a specific *fruit* and therefore an *apple cake* is a specific *fruit cake*. If the modifiers are unrelated (or in a part-of relation), we cannot come to any conclusion, though. For instance, there is no obvious relation between a *kitchen chair* and a *database chair*, as the two words *kitchen* and *database* are unrelated.

Before we start our query, we thus check if both words appear in the repository. In case that they do not appear, we use the presented techniques of compound preprocessing. With this, we can handle many more terms than the ones that appear in our repository.

4.2 Finding Paths

The memory-based implementation of SemRep allows for a very fast determination of paths between concepts. We use the following procedure to determine all directed paths between two concepts a, b for up to length 4:

1. We first iterate through all direct relations of a. If we find b as one target concept, we found a path of length 1.
2. For paths of length 2, we determine for both a and b all outgoing paths of length 1 and the target concepts, denoted as $T(a), T(b)$. Then we determine all concepts c for which holds $c \in T(a), c \in T(b)$ to return a path of length 2 between a and b.
3. For length 3, we determine for a the target concepts of all outgoing paths of length 2, denoted as $T'(a)$. Similarly as in the step before, we now look for all concepts c for which holds $c \in T'(a), c \in T(b)$. This way, we obtain paths of length 3 between a and b.
4. Finally, to determine paths of length 4 we calculate $T'(b)$ and determine all concepts c for which holds $c \in T'(b), c \in T(a) \cup T'(a)$.

Thus, in the worst case we calculate all paths of length 2 for both concepts which is much faster than calculating all paths of length 4 as a default breadth-first algorithm would do. Each path that was found is checked on its validity, i.e., whether we can determine a reasonable semantic relation type according to the approach discussed in the next subsection.

We support different configurations regarding the path lengths to evaluate. The algorithm can stop as soon as at least one path is found, which we call *First Path*, or it can run all steps in order to determine the most reliable path (we call this *All Paths*). Both configurations can retrieve multiple paths for a single query, but differ in quality and processing time. According to our experiences, *All Paths* allows rarely better results than *First Path*, because longer paths are usually less reliable than shorter ones but cause much longer execution times. We will attend to this issue in more detail in the evaluation section.

4.3 Determining the Path Type

For paths of length 2 or more it is necessary to determine what semantic relation holds between concepts a and b and how reliable this path is. The goal is to return the relation type of the path with the highest confidence, though we could offer several candidates for manual selection by a user.

Let P be a path of length = 2, consisting of three concepts c_1, c_2, c_3 with two relations r_1, r_2 in between. The path type of the combined path is called r. To determine r from r_1 and r_2 we have to consider multiple cases and Fig. 3 shows which resulting path type we choose for SemRep. As one can see and we discuss below, in some cases we derive a new, weaker relation type $related$ and in some cases we cannot derive a semantic relation type. Relation type $related$ is not stored in SemRep but can only be derived as the result of query processing.

c1 – c2 \ c2 – c3	equal	is-a	inv. is-a	part-of	has-a
equal	*equal*	*is-a*	*inv. is-a*	*part-of*	*has-a*
is-a	*is-a*	*is-a*	*related*	*part-of*	*has-a*
inv. is-a	*inv. is-a*	–	*inv. is-a*	*part-of*	*has-a*
part-of	*part-of*	*part-of*	*part-of*	*part-of*	*related*
has-a	*has-a*	*has-a*	*has-a*	–	*has-a*

Figure 3: Relation type resulting from complex paths between concepts c_1, c_2, c_3.

In the following discussion we distinguish three main cases for deriving the combined relation type for paths of length 2 where $f^{-1}(r)$ denotes the inverse of a relation type r:

1. $r_1 = r_2$ (homogeneous path)
2. $r_1 \neq r_2, r_1 \neq f^{-1}(r_2)$ (heterogenous, non-inverse path)
3. $r_1 \neq r_2, r_1 = f^{-1}(r_2)$ (heterogenous, inverse path)

Homogeneous paths

Handling homogeneous paths is easy. Since all relation types are assumed to be transitive, it simply holds $r = r_1 = r_2$. We may, however, encounter the problem that c_2 is a homonym which can lead to a wrong relation. For instance, the homonym *table* could lead to the path *desk is-a table is-a data structure*, which would lead to the false relation *desk is-a data structure*. SemRep has not yet a solution to deal with such homonym problems since we only capture concept names. Dealing with homonyms is a general problem in linguistic resources and matching that needs more attention. One possible approach for SemRep could be to maintain several concepts for homonyms and differentiate them by a domain specification or other context information.

Heterogeneous, non-inverse paths

We made a significant observations for heterogeneous paths: The relation types have different binding strengths. Equality has the least binding strength, followed by is-a and followed by part-of, which has the highest binding strength. Thus, the relation type of the highest binding strength in the path determines the overall path type and Fig. 3 shows all possible combinations.

For example, in *engine part-of car is-a motor vehicle* the relation *engine part-of motor vehicle* holds, because part-of dominates against is-a. Quite similarly, in *Computer has-a RAM is-a memory* the relation *Computer has-a memory* holds.

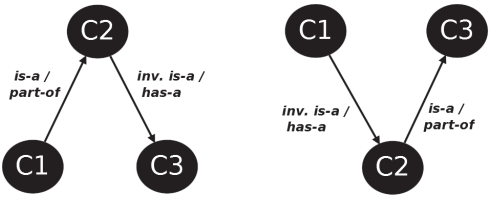


Figure 4: The two different kinds of inverse paths: Paths where a semantic relation can be derived (left) and where it cannot be derived (right).

Heterogeneous, inverse paths

There are 4 inverse combinations, is-a + inv. is-a, part-of + has-a and the opposite cases. These cases need special treatment.

1. $\langle c_1$ is-a c_2 inv. is-a $c_3\rangle$: In this case the relation type *related* is derived. Both c_1 and c_3 have a common father concept c_2, i.e., they share some attributes, but also differ in some attributes. A typical example is *apple – fruit – banana* (see also Fig. 4, left).

2. $\langle c_1$ inv. is-a c_2 is-a $c_3\rangle$ In this case, we have a completely different situation, though it is just the inverse path of case 1 (see also Fig. 4, right). Now, c_2 is a subconcept of c_1 but then the path leads to a different concept c_3. The concept c_2 shares attributes

with both c_1 and c_3, but it is unclear whether c_1 and c_3 are related or share any attributes. Thus, in this case it is impossible to derive a specific relation type. Examples can be *fruit – apple – plant tissue* (is-a), *fruit – apple – tree fruits* (inv. is-a) and *fruit – apple – fructus* (equal), showing that many relation types can actually be possible.

3. $\langle c_1$ part-of c_2 has-a $c_3\rangle$ This case is similar to case 1, but (c_1, c_2) and (c_2, c_3) do not share any attributes here. The only way c_1 and c_3 are related is their co-occurrence in c_2. We can also derive the type *related* here, though it may not always be sensible. Some examples are *CPU – Computer – Memory* (so memory and CPU are related) and *roof – house – cellar* (roof and cellar are related).

4. $\langle c_1$ has-a c_2 part-of $c_3\rangle$ This last case is similar to case 2 and no relation can be unequivocally derived. Examples include *Laptop – CPU – Notebook* (equal) and *Laptop – CPU – Computer* (is-a) and *Laptop – CPU – Netbook* (inv. is-a).

Therefore, we will not return any relation type in cases 2 and 4 and instead return *unrelated*.

Paths of length 3 or more

Paths of length 3 or more can be dealt with by combining the first two relation types according to Fig. 3 and combining the resulting relation type step by step with further relation types. Consider the example *combustion engine is-a engine part-of car inv. is-a convertible*. The first two relations (is-a + part-of) lead to *combustion engine part-of car*. To complete the path, we have to resolve *combustion engine part-of car inv. is-a convertible*. According to Fig. 3 it holds *combustion engine part-of convertible*.

For the result type $related$ it is not meaningful to combine it with further relations; hence we have not considered it as a possible input type in Fig. 3.

4.4 Confidence Calculation

As we may have different paths suggesting different relation types for a specific query, we need a measure to score paths and thus to be able to find the most likely type to hold.

In our scoring function, we consider three parameters:

1. The resource of each relation within the path
2. The relation type of each relation within the path
3. The path length

The rationale behind these parameters is that relations from manually curated resources (like WordNet) are considered as more reliable than automatically generated ones and that

Parameter	Value
c(r) = WordNet	0.96
c(r) = Wikipedia	0.90
c(r) = UMLS	0.98
c(r) = OpenThesaurus	0.95
c(t) = equal	0.98
c(t) = is-a / inv. is-a	0.96
c(t) = has-a / part-of	0.8
INV	0.15

Table 3: Values for the types and resource confidences used in formula (1).

some relation types are generally more reliable (is-a) than others (part-of). Finally, the longer a path becomes, the more it is prone to errors.

Let r be a relation within the path, $c(r)$ be the confidence of the resource where this relation comes from and $c(t)$ the confidence of the specific relation type of r ($c(r), c(t) \in [0, 1]$). Our scoring function is:

$$s = c(r_1) * c(t_1) \ * \ c(r_2) * c(t_2) \ * \ ... \ * \ c(r_n) * c(t_n) \ - \ INV \quad (1)$$

In formula (1), INV is a parameter that is used if we have a inverse path ("related" type).

Currently, we set the configuration shown in Table 3, which proved to lead to the best results in our experiments. The values also reflect the rationale outlined above by giving Wikipedia a slightly lower confidence than the manually curated resources WordNet or UMLS.

4.5 Handling Empty Results

In some cases, the repository cannot find any path from source to target concept. We discovered this in the example *(wine region, location)*, which is actually a simple is-a correspondence. If the concept *wine region* would not be contained by the repository, the preprocessor would immediately reduce it to the word *region* (GMR) and the repository would compare *region* with *location*. There is indeed an is-a relation between those concepts expressed in the repository. However, the preproecessor is only ran if at least one of the two input concepts is not contained by the dictionary, so GMR is not applied in this case as it does not seem necessary.

To further improve the recall, we perform a so-called Post GMR. If no relation is found and one of the two concepts is a compound word, just as in the above example, we apply GMR on this compound word and run the query executor again. With this, we can correctly resolve correspondences like *(wine region, location)*, even though there is no path between the two.

Benchmark	Src. Nodes	Trg. Nodes	# Corr.	equal	is-a	part-of	rel.
Furniture	25	174	136	15	111	10	0
Groceries	58	333	169	32	127	2	8
Clothing	31	153	144	5	130	8	0

Table 4: Overview of the evaluation scenarios and benchmark mappings.

5 Evaluation

We evaluate the use of the introduced SemRep repository for three semantic ontology matching tasks from different domains. The matching tasks are addressed by the tool STROMA that so far used only manually curated knowledge resources such as WordNet. After describing the data sets and experimental setup we present the obtained F-measure results using SemRep, as well as the query execution times. We also provide a comparative evaluation of using STROMA without and with SemRep.

5.1 Experimental Setup

We evaluate the effectiveness of using the SemRep repository for three semantic ontology match tasks called Furniture (F), Groceries (G) and Clothing (C). Benchmarks F and G match parts of the category tree of Amazon and Ebay while benchmark C matches between Amazon and Zalando. We manually generated a perfect mapping for each benchmark together with the semantic relation type per correspondence. Table 4 gives an overview of the three benchmarks, together with the number of correspondences and specific correspondence types they contain.

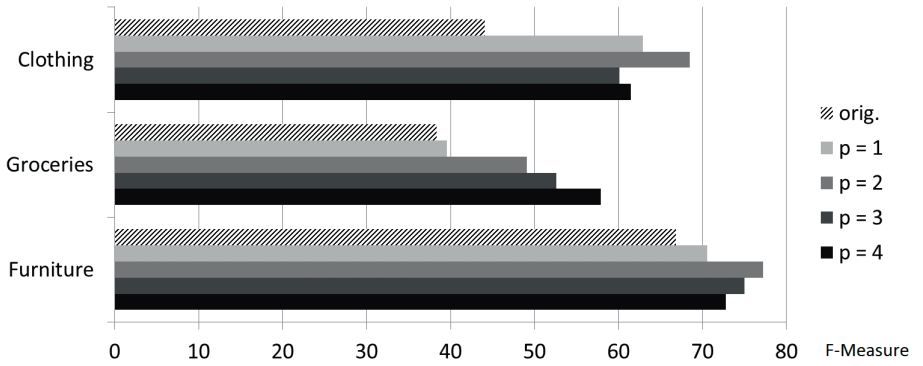
For ontology matching we use our tool STROMA [AR14a] which applies a so-called enrichment strategy. That means it enriches the mapping result obtained by a standard ontology match tool that only determines untyped correspondences in a post-processing step to determine the semantic relation type per correspondence. For this purpose it applies several linguistic and structural strategies and utilizes background knowledge resources such as WordNet or UMLS (for the benchmarks only WordNet is relevant). It also applies a so-called Multiple Linkage strategy, which automatically assigns the is-a relation type to correspondences that refer to a node where many correspondences point to ([AR14a]). We decided to deactivate this function in our evaluation since it infers with the use of the background knowledge resources, especially since the ontologies in the benchmarks differ quite substantially in size.

SemRep is executed on a Windows Server 2008 with 72 GB RAM and an Intel XEON 2.53 GHz processor (30 GB were reserved for the system). We use the configuration *First Path*.

5.2 F-Measure and Execution Times

We first evaluate the F-Measure results for semantic matching with STROMA using SemRep as a background knowledge resource to look up the semantic relation type between pairs of concepts occurring in the input ontologies. Fig. 5 shows the obtained F-measure results for the three match tasks without using SemRep ("'orig."') and using SemRep for different maximal path lengths p (1-4) to identify a semantic relation type between two concepts of interest. While the results are somewhat heterogeneous, we observe that the use of SemRep improves match quality in all cases. For two of the three benchmarks the best F-Measure is achieved for p=2; for the Clothing match task F-Measure improves from 43% to 68%, which is an excellent outcome. Considering path lengths larger than 2 can help to improve recall, but at the expense of higher execution times (see below). This effort payed off only for the test case G (Groceries) which had the poorest F-Measure without using SemRep.

Figure 5: F-Measure for the 3 scenarios for the different path lengths 1 to 4. The column *orig.* shows the original STROMA results without using the repository.



Next we evaluate the use of SemRep w.r.t. to the minimum acceptance threshold the confidence of a determined semantic relation type must meet to be returned. Table 5 shows the results for acceptance thresholds between 0.4 and 0.8 and for maximal path lengths of 2 and 3. It can be seen that for p=3 the results decrease for thresholds higher than 0.6 and that no better results can be obtained than for p=2 (except for the groceries benchmark). Since the minimal score of a path of length 2 is $0.8 * 0.8 = 0.64$, the results remain the same for thresholds below 0.6. The experiments suggest that a threshold of 0.6 is a good default setting as well as a maximal path length of 2.

Table 6 shows the execution times for the different benchmark scenarios. Table 6a shows the execution times for each benchmark in seconds, while Table 6b shows the average execution time for one correspondence. If paths until a maximum length of 2 are used, the execution time is very fast (2.2 to 4.9 ms per query). However, as soon as a maximum path length of 3 is allowed, complex paths have to be calculated and the execution time

T	p = 2			p = 3		
	F	G	C	F	G	C
0.8	67.6	43.7	66.4	72.1	47.9	58.0
0.7	74.3	47.3	67.1	74.3	48.5	58.7
0.6	77.2	49.1	68.5	75.7	50.9	58.7
0.5	77.2	49.1	68.5	75.0	53.2	59.4
0.4	77.2	49.1	68.5	74.2	53.2	60.1

Table 5: F-measure for path lengths of 2 and 3 and different acceptance thresholds T.

	p=1	p=2	p=3	p=4
F	0.33	0.37	26	39
G	0.37	0.44	19	22
C	0.65	0.7	130	136

(a) Total execution times for each benchmark.

	p=1	p=2	p=3	p=4
F	2.4	2.7	191	287
G	2.2	2.6	112	130
C	4.5	4.9	902	944

(b) Average execution times per correspondence.

Table 6: Execution times for the three benchmarks using different maximal path lengths (1 to 4).

increases considerably. Maximum path lengths of 4 increase execution times further, but only to a little degree. We assume this is because most paths retrieved by the repository are of length 3 or less.

Initially we also experimented with a standard breadth-first search. However it took several times longer that the optimized approach, e.g. already 8 seconds for evaluating paths of maximal length 3.

5.3 Relative Quality Impact of Using SemRep

In the last experiment, we compare the effectiveness of different STROMA configurations without and with using SemRep to determine the impact of the new repository on the match quality. We use two different configurations of STROMA: Configuration 1 (C1) is the default configuration where all implemented strategies are enabled, i.e., both the background knowledge strategy and generic strategies like linguistic or structural techniques. In configuration 2 (C2), all generic strategies are disabled so that the match quality is solely determined by the used repository for background knowledge (WordNet or SemRep). We assume that this configuration can better illustrate the relative usefulness of the repositories, while the first configuration provides their overall impact on mapping quality. For SemRep we apply the default strategy with p=2.

Table 7 shows the F-Measure results and total runtimes for the three benchmarks. Columns C_1 for both the original STROMA with WordNet and the new version with SemRep show that SemRep allows for a substantial overall mapping improvement between 10.7 and 24.4

	Original results			New results		
	C_1	C_2	time	C_1	C_2	time
F	66.9	23.5	4.33 s	82.3	44.9	0.37 s
G	38.4	27.2	4.73 s	48.5	39.1	0.44 s
C	44.1	11.8	2.41 s	72.0	44.8	0.7 s

Table 7: F-measure and execution times for the three mappings using STROMA, for the original tool without the repository (left) and for the new tool using the repository (right).

%. The relative quality of the repository becomes even clearer with the results in columns C_2. While in the original STROMA using only WordNet, only about 12 to 27 % F-measure could be reached, the use of SemRep alone allows for F-measure values between about 43 and 53 %. The difference is especially visible for benchmark G, where SemRep alone (42.6 %) achieves better results than the original STROMA running all strategies (38.4 %). The boost in F-measure is mostly caused by the additional knowledge from Wikipedia, which provides relations to many more (and more specific) relations, and the combination of Wikipedia and WordNet relations.

In addition to the quality improvements, execution times were considerably reduced, now being constantly below one second. This improvement is caused by keeping the data in main memory, which was not the case in the previous implementation where we used an API to access WordNet data locally. As a result, using SemRep provides significant improvements both in quality and performance.

6 Outlook and Future Work

We presented SemRep, a new repository infrastructure to integrate different dictionaries and thesauri to support schema and ontology mapping tasks. We successfully imported more than a million concepts and around 4.5 million relations. SemRep allows a fast lookup of semantic concept relations and the combination of several relations, possibly of different types, to improve the coverage and to interconnect knowledge from different sources.

There are still several issues and opportunities for improving SemRep and its applications. First, we could invest more to check and improve the quality of automatically extracted semantic relations. Second, we need to better deal with homonyms in order to limit the derivation of wrong semantic relations. We may also support more user interaction to check the validity of derived semantic relations and propagate back corrections into the repository.

7 Acknowledgments

This study was partly funded by the European Commission through Project "LinkedDesign" (No. 284613 FoF-ICT-2011.7.4).

References

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, ISWC/ASWC, pages 722–735, Berlin, Heidelberg, 2007. Springer-Verlag.

[AKtKvH06] Z. Aleksovski, M. Klein, W. ten Kate, and F. van Harmelen. Matching Unstructured Vocabularies using a Background Ontology. In S. Staab and V. Svatek, editors, *Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, number 4248 in Lecture Notes in Artificial Intelligence, pages 182–197. Springer, 2006.

[AR13] Patrick Arnold and Erhard Rahm. Semantic Enrichment of Ontology Mappings: A Linguistic-Based Approach. In *Advances in Databases and Information Systems*, volume 8133, pages 42–55. Springer, 2013.

[AR14a] Patrick Arnold and Erhard Rahm. Enriching Ontology Mappings with Semantic Relations. *Data and Knowledge Engineering*, 2014.

[AR14b] Patrick Arnold and Erhard Rahm. Extracting Semantic Concept Relations from Wikipedia. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics*, pages 26:1–26:11, New York, NY, USA, 2014. ACM.

[BBR11] Z. Bellahsene., A. Bonifati, and E. Rahm. *Schema Matching and Mapping*. Springer, 2011.

[BEP+08] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1247–1250, New York, NY, USA, 2008. ACM.

[DR02] Hong Hai Do and Erhard Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proc. 28th Intl. Conference on Very Large Databases (VLDB), Hongkong*, 2002.

[EKGH+12] Judith Eckle-Kohler, Iryna Gurevych, Silvana Hartmann, Michael Matuschek, and Christian M. Meyer. UBY-LMF - A Uniform Model for Standardizing Heterogeneous Lexical-Semantic Resources in ISO-LMF. In *Proceedings of the 8th International Conference on Language Resources and Evaluation (LREC)*, pages 275–282, 5 2012.

[GAP12] Fausto Giunchiglia, Aliaksandr Autayeu, and Juan Pane. S-match: An Open Source Framework for Matching Lightweight Ontologies. *Semantic Web*, 3(3):307–317, August 2012.

[GHKR11] A. Gross, M. Hartung, T. Kirsten, and E. Rahm. Mapping Composition for Matching Large Life Science Ontologies. In *2nd Intl. Conference on Biomedical Ontology (ICBO)*, 2011.

[GM98] Christian Fellbaum George Miller. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[MBDH05] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Halevy. Corpus-Based Schema Matching. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 57–68, Washington, DC, USA, 2005. IEEE Computer Society.

[NP10] Roberto Navigli and Simone Paolo Ponzetto. BabelNet: Building a Very Large Multilingual Semantic Network. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL, pages 216–225, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[RS07] Chantal Reynaud and Brigitte Safar. Exploiting WordNet as Background Knowledge. In *International ISWC Ontology Matching Workshop*, 2007.

[SdM06] Marta Sabou, Mathieu d'Aquin, and Enrico Motta. Using the Semantic Web as Background Knowledge for Ontology Mapping. In *Ontology Matching*, 2006.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, pages 697–706, New York, NY, USA, 2007. ACM.

[WLWZ12] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q. Zhu. Probase: A Probabilistic Taxonomy for Text Understanding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, New York, NY, USA, 2012. ACM.

Physical Layer

Energy and Performance—Can a Wimpy-Node Cluster Challenge a Brawny Server?

Daniel Schall, Theo Härder
{schall, haerder}@cs.uni-kl.de
DBIS Group, University of Kaiserslautern, Germany

Abstract: Traditional DBMS servers are often over-provisioned for most of their daily workloads and, because they do not provide energy proportionality, waste more energy than necessary. A cluster of wimpy servers, where the number of nodes can dynamically adjust to the current workload, might offer better energy characteristics for these workloads. Yet, clusters suffer from *friction losses* and cannot quickly adapt to the workload, whereas a single server delivers maximum performance instantaneously.

Designed for a cluster of nodes, our WattDB system primarily aims at energy proportionality for a wide range of DB applications. In this paper, we check this system under OLTP and OLAP workloads against a single-server DBMS in terms of throughput/response time and energy efficiency. To test the system's ability to adjust to changing workloads, we execute several benchmark at differing system activity levels. To quantify possible energy saving and its conceivable drawback on query runtime, we evaluate our WattDB implementation—to obtain maximum accuracy possible—on a cluster of wimpy nodes as well as on a single, brawny server and compare the results w.r.t. performance and energy consumption. Our findings confirm that—especially for OLAP workloads—energy can be saved without sacrificing too much performance.

1 Introduction

Saving energy is a concern in all areas of IT. Studies have shown that single servers have only limited potential for energy optimizations and that, in general, the best performing configuration is also the most energy-efficient one [THS10]. This is due to the narrow power spectrum between idle and full utilization of a single server [BH07].

Today's server hardware is not energy-proportional; even at low utilization levels, it consumes a significant amount of power rapidly approaching peak power [HHOS11]. Scaling systems down automatically when idle, thus preventing high idle power consumption, is the main focus of today's servers to achieve energy efficiency. Of course, several components such as CPUs are able to quickly change into sleep states, requiring less energy, when idle. But other components, such as two main energy consumers of DBMSs—main memory and storage drives—exhibit bad and more or less load-independent energy characteristics. Hence, especially for DB servers, ~50% of its power is already wasted when idle.

Therefore, better energy proportionality cannot be achieved with current, centralized solutions. This observation also holds for traditional DBMSs, composed of a single server with

huge main memory and lots of storage drives attached. In contrast to centralized, brawny servers, a scale-out cluster of lightweight (wimpy) servers has the ability to shutdown single nodes independently. At an abstract level, this enables the cluster to dynamically add or remove storage and processing power based on the cluster's utilization.

With cloud computing, elastic systems have emerged that adapt their size to the current workload. While stateless or lightweight systems can easily increase or reduce the number of active computing nodes in a cluster, a database faces much more challenges due to the need of high interactions among the nodes and fast reachability of DB data. Similar to cloud-based solutions, we hypothesize that a cluster of nodes may adjust the number of active nodes to the current demand and, thus, approximate energy proportionality.

Based on these observations, we developed WattDB, a research prototype of a distributed DBMS cluster, running on lightweight, Amdahl-balanced nodes using commodity hardware. The cluster is intended to dynamically shrink and expand its size, dependent on the workload. However, reconfiguring a cluster to balance node utilization requires data to be moved from node to node. But, copying data is time-consuming and adds overhead to the already loaded cluster. Hence reducing both, time and overhead, is crucial for an elastic DBMS. Although the cluster may not be as powerful as a monolithic server, we expect our system to consume significantly less energy for typical DBMS workloads.

In this paper, we compare a single, brawny server with a cluster of wimpy nodes under OLTP and OLAP workloads, running TPC-H and TPC-C benchmarks, respectively. In Sect. 2, we give an overview of recent research addressing *partitioning, elasticity, and energy efficiency* of DBMSs. Sect. 3 introduces important aspects of our energy-proportional database cluster, whereas Sect. 4 contains the results of several empirical experiments and compares energy use and performance of our cluster to those of a brawny server. In Sect. 5, we summarize the main issues of our work and give some conclusions.

2 Related Work

Reducing energy consumption of servers and enabling dynamic reconfiguration are all subject to a variety of research approaches. For the reason, we give a short overview of related works in three fields that serve as a building blocks of our research.

2.1 Dynamic Clustering

Traditional clustered DBMSs do not dynamically adjust their size (in terms of the number of active nodes) to their workload. Hence, scale-out to additional nodes is typically supported, whereas the opposite functionality, shrinking the cluster and centralizing the processing—the so-called scale-in—, is not. Recently, with the emergence of clouds, a change of thinking occurred and dynamic solutions became a research topic.

In his PhD thesis [Das11], Sudipto Das implemented an elastic data storage, called Elastras, able to dynamically grow and shrink on a cloud. As common in generic clouds,

his work is based on decoupled storage where all I/O involves network communication. *Key Groups*, application-defined sets of records frequently accessed together, can be seen as dynamic partitions that are often formed and dissolved. By distributing the partitions among nodes in the cluster, both performance and cost can be controlled.

A lot more data management systems working on a cloud have been proposed. In [BFG+08], Brantner et al. designed a DBMS using Amazon S3 as storage and running on top. Lomet et al. [LFWZ09] divided the database into two layers, one transactional and one persistence component that can run independently.

In [AFP+09], Armbrust et al. proposed a scalable storage layer supporting consistency and dynamic scale-out/in called SCADS. Objects in SCADS are stored in logical order. Hot, i. e., frequently accessed objects are distributed among disks to improve access latencies and mitigate bottlenecks. The system was also extended to automatically adjust to workload changes and autonomously redistribute data.

Besides relational approaches, other implementations relax traditional DBMS properties to gain performance and simplify partitioning. Yahoo PNUTS [CRS+08], Bigtable [CDG+08], and Cassandra[1], are example of systems sacrificing transaction or schema support and query power [AFP+09]. Instead of arbitrary access patterns on the data, only primary key accesses to a single record are supported [VCO10].

Amazon's SimpleDB[2] allows transactions to access multiple records, but limits accesses to single tables. Moreover, most current scalable data storage systems lack the rich data model of an RDBMS, which burdens application developers with data management tasks. Yet, no *fully-autonomous, clustered DBMS* exists providing ACID properties for transactions and SQL-like queries while dynamically adjusting its size to the current workload.

2.2 Energy Optimizations

Lang et al. [LHP+12] have shown that a cluster suffers from friction losses due to coordination and data shipping overhead and is therefore not as powerful as a comparable heavyweight server. On the other hand, for moderate workloads, i. e., the majority of real-world database applications, a scale-out cluster can exploit its ability to reduce or increase its size sufficiently fast and, in turn, gain far better energy efficiency.

In [SH13a], we already explored the capabilities and limitations of a clustered storage architecture that dynamically adjusts the number of nodes to varying workloads consisting of simple *read-only page requests* where a large file had to be accessed via an index. We concluded that it is possible to approximate energy proportionality in the storage layer with a cluster of wimpy nodes. However, attaching or detaching a storage server is rather expensive, because (parts of) datasets may have to be migrated. Therefore, such events (in appropriate workloads) should happen on a scale of minutes or hours, but not seconds.

In [SH13b], we focused on the query processing layer—again for varying workloads consisting of two types of *read-only SQL queries*—with similar conclusions. In this contribu-

[1] `http://cassandra.apache.org/`
[2] `http://aws.amazon.com/simpledb/`

tion, we revealed that attaching or detaching a (pure) processing node is rather inexpensive, because repartitioning and movement of data is not needed. Hence, such an event can happen in the range of a few seconds—without disturbing the current workload too much.

We substantially extended the kind of DBMS processing supported by WattDB to *complex OLAP / OLTP workloads consisting of read-write transactions* in [SH14]. For this purpose, we refined and combined both approaches to get one step closer to a fully-featured DBMS, able to process OLTP and OLAP workloads simultaneously. In this work, we were able to trade performance for energy savings and vice versa. Yet, we found out that the adaptation of the cluster and the data distribution to fit the query workload is time-consuming and needs to be optimized.

As discovered before, a single-server-based DBMS is far from being energy-proportional and cannot process realistic workloads in an energy-efficient way. Our previous research indicates that a cluster of lightweight (wimpy) servers, where nodes can be dynamically switched on or off, seems more promising. In this paper, we compare our scale-out cluster to a big server to quantify possible energy savings and to discover promising workloads.

3 Cluster vs. Big Server

Our cluster hardware consists of n (currently 10) identical nodes, interconnected by a Gigabit-ethernet switch. Each node is equipped with an Intel Atom D510 CPU (with two threads using HyperThreading) running at 1.66 GHz, 2 GB of DRAM and an SSD for data storage. The configuration is considered Amdahl-balanced [SBH+10], i. e., balanced w.r.t. I/O and network throughput on one hand and processing power on the other. By choosing commodity hardware with limited data bandwidth, Gigabit-Ethernet wiring is sufficient for interconnecting the nodes. All nodes can communicate directly.

To compare performance and energy savings, we ran the same experiments again on a single, brawny server. This server has two *Intel Xeon X5670* processors with 24 GB of RAM and 10 SSDs.[3] Each CPU has 12 cores and 24 threads (using HyperThreading), running at 2.93 GHz.

Fig. 1 sketches the cluster with 10 nodes and the big server. For comparison, we have highlighted the main components (CPU cores, main memory and disk) inside the nodes as well as the communication network. Each wimpy node consumes ∼22 – 26 Watts when active (based on utilization) and ∼2.5 Watts in standby. The interconnecting network switch consumes ∼20 Watts and is included in all measurements.

In its minimal configuration—with only one node and the switch running and all other nodes in standby—the cluster consumes ∼65 Watts. This configuration does not include any disk drives, hence, a more realistic minimal configuration requires ∼70 Watts. In this state, a single node is serving the entire DBMS functionality (storage, processing, and cluster coordination). With all nodes running at full utilization, the cluster will consume

[3] For a fair comparison with the cluster of wimpy nodes, we reduced RAM to 24GB, although the server can handle much more. Yet, with more main memory, the power use of the server would also be much higher.

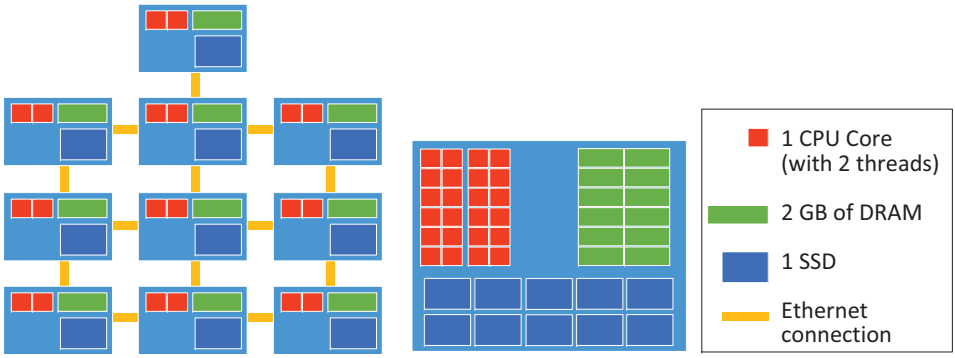


Figure 1: The 10-node cluster compared with the brawny server

~260 to 280 Watts, depending on the number of disk drives installed. This is another reason for choosing commodity hardware which uses much less energy compared to server-grade components. For example, main memory consumes ~2.5 Watts per DIMM module, whereas ECC memory, used in the brawny server, consumes ~10 Watts per DIMM.

The power consumption of the big server (with 10 SSDs) ranges from ~200 Watts when idle to ~430 Watts at full utilization.[4] In theory, the systems should show similar performance. All nodes in the cluster come with 16.6 (10x1.66) GFLOPS, whereas the performance of the big server is rated with 17.6 GFLOPS. Furthermore, L2 caches and memory bandwidth of both systems are similar and the same number of disks is installed.

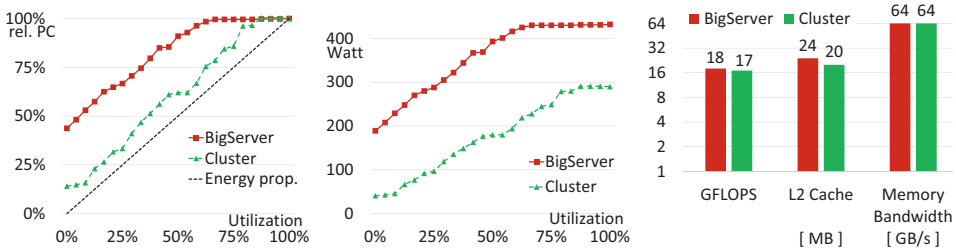
Fig. 2 visualizes power consumption and theoretical performance figures, as given in the product sheets, for both systems. Of course, the relationships shown for the power consumption are idealized, because power consumption of the big server and the full cluster may differ to a certain amount. Fig. 2a shows how energy proportionality drawn for the big server is much better approximated by the cluster where at least 1 and—depending on the workload—up to 10 nodes are active. Fig. 2b gives an impression about the power consumption of both systems when running at a specific activity level.

3.1 DBMS Software

By the time, research gained interest in energy efficiency of database servers, no state-of-the-art DBMS was able to run on a dynamically adapting cluster. To test our hypothesis (see Sect. 1), we developed *WattDB* that supports SQL query processing with ACID properties, but is also able to adjust to the workload by scaling out or in, respectively.

To enable a fair comparison, the same software is running on the cluster and the big server. On the latter, the dynamic features of WattDB are not needed and are therefore disabled. The smallest configuration of WattDB is a single server, hosting all database functions and

[4]These measurements include only 24 GB of DRAM as previously explained.



(a) Relative power consum. (b) Absolute power consum. (c) Performance figures

Figure 2: Power consumption and theoretical performance figures for both systems

acting as endpoint to DB clients. This server is called *master node*. DB objects (tables, partitions) and query evaluation can be offloaded to arbitrary nodes in the cluster to relieve the node, but it will always act as the coordinator and client endpoint. Some of the key features and design considerations of WattDB are explained in the following.

3.2 Dynamic Query Processing

To run queries on a cluster of nodes, distributed query plans are generated on the master node. Except data access operators which need local access to the database's records, all query operators can be placed on remote nodes. Running query operators on a single node does not involve network communication among query operators, because all records are transferred via main memory. Distributing operators implies shipping of records among nodes and, hence, introduces network latencies. Additionally, the bandwidth of the Gigabit Ethernet, which we are using for our experiments, is relatively small, compared to memory bandwidth.

To mitigate the negative effects of distribution, WattDB is using *vectorized volcano-style query operators* [Gra94, BZN05]; hence, operators ship a set of records on each call. This reduces the number of calls between operators and, thus, network latencies. To further decrease network latencies, buffering operators are used to prefetch records from remote nodes. *Buffering operators* act as proxies between two (regular) operators; they asynchronously prefetch sets of records, thus, hiding the delay of fetching the next bunch of records.

In WattDB, the query optimizer tries to put pipelining operators[5] on the same node to minimize latencies. Offloading pipeline operators to a remote node has little effect on workload balancing and, thus, does not pay off. Instead, blocking operators[6] may be placed on remote nodes to equally distribute query processing. They generally consume more resources (CPU, main memory) and are therefore prime candidates for workload balancing in the cluster.

[5]Pipelining operators can process one record at a time and emit the result, e. g., projection operators.
[6]Blocking operators need to receive all records before emitting the first result record, e. g., sorting operators.

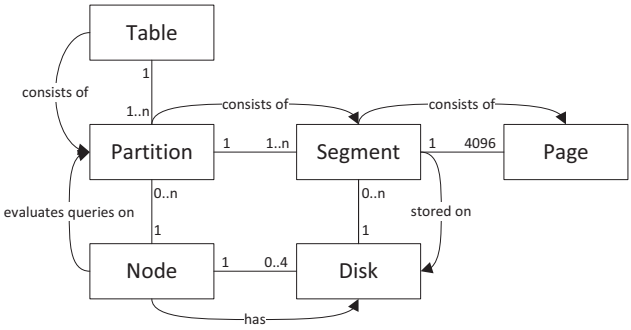


Figure 3: Database schema

3.3 Dynamic Reorganization

The master node is coordinating the whole cluster. It is globally optimizing the query plans, whereas regular nodes can locally optimize their part of the plan. Furthermore, it takes nodes on- and offline and decides when and how DB tables are (re)partitioned.

Every node is monitoring its utilization: CPU, memory consumption, network I/O, and disk utilization (storage and IOPS). Additionally, performance-critical data is collected for each database partition, i. e., CPU cycles, buffer page requests and network I/O. With these figures, we can correlate the observed utilization of cluster components to (logical) database entities. Hence, both types of data are necessary to identify sources of cluster imbalance. We use the performance figures of the components to identify their over- or under-utilization. In addition, activity recording of database entities is needed to determine the origin of the cluster's imbalance. For this reason, the nodes send their recording every few seconds to the master node.

The master checks the incoming performance data to predefined thresholds—with both upper and lower bounds. If an overloaded component is detected, it will decide where to distribute data and whether to power on additional nodes and resume their cluster participation. Similar, underutilized nodes trigger a scale-in protocol, i. e., the master will distribute the data (processing) to fewer nodes and shutdown the nodes currently not needed. Decisions, what data to migrate and where, are done based on the current utilization of the nodes, the expected query workload, and the estimated cost, it will take to migrate data between nodes.

In WattDB, we have implemented different policies regarding the scale-out behavior. First, each node in the cluster stores data on local disks to minimize network communication. If storage space of a node is in short supply, database partitions are split up on nodes with free space.

Second, WattDB tries to keep the I/O rate for each storage disk in a certain range. Underutilized disks are eligible for additional data—either newly generated by INSERT operations or migrated from overloaded disks. Utilization among storage disks is first locally balanced on each node, before an allocation of data from/to other nodes is considered.

Third, each node's CPU utilization should not exceed the upper bound of the specified threshold (80%). As soon as this bound is violated on a node, for more than one minute, WattDB first tries to offload query processing to underutilized nodes.[7] If the overload situation cannot be resolved by redistributing the query load, the current data partitions and their node assignments are reconsidered. When a partition causing overload is identified, it is split according the partitioning scheme applied, where affected segments are moved to other nodes [SH15]. Fig. 3 sketches the relationships between tables, partitions, nodes, and segments.

For underutilized nodes, an inverse approach is needed. A scale-in protocol is initiated, which quiesces the involved nodes from query processing and shifts their data partitions to nodes currently having sufficient processing capacity, as soon as the average utilization over one minute falls below 30%.[8]

Similar rules exist for network and memory utilization, e. g., if the working sets of the transactions become too big for the DB buffer, repartitioning is triggered. WattDB makes decisions based on the current workload, the course of utilization in the recent past, and the expected future workloads [KHH12]. Additionally, workload shifts can be user-defined to inform the cluster of an expected change in utilization.

Cost of reorganization Moving data is an expensive task, in terms of energy consumption and performance impact on concurrently running queries. Data reorganization binds some computing resources, which would be needed to optimally process the query workload. This competition leads to fewer resources for running database workloads and, in turn, reduces query throughput. However, the reorganization cost should amortize by reducing the energy consumption of subsequent queries. Though it is difficult to calculate the exact energy consumption of a data move operation with respect to the impact of running queries, the energy cost can be estimated with the duration of the move operation and the (additional) power consumption. Hence, moving 1 GByte of data to a dedicated node with 25 Watts power consumption will require approximately 10 seconds and 250 Joules.

To save energy, reconfiguration overhead needs to pay off by reducing future query runtimes. Likewise, scale-in must trigger when the cluster is able to handle the workload with less nodes. To estimate the impact of reorganization, WattDB relies on a simplified cost model where upcoming workload predictions and maintenance costs are calculated.

3.4 Power Measurement

We have developed a measurement device, capable of monitoring the power and energy consumption of each node in the cluster. The device is also able to monitor these metrics of the big server. This device sends the stream of measurements to a connected PC, running the monitoring software. The monitoring software can further capture the number of active database nodes and the total throughput and response times of queries during the tests.

[7] This works well for operators like **SORT**, **GROUP**, and **AGGREGATE**.

[8] The rules implemented are a little more sophisticated, depending on additional factors, e. g., overall utilization and intensity of workload shifts. In this work, we illustrate a simplified model to keep things short.

This computer is controlling the benchmark execution by submitting queries to the master node; thus, it enables fine-grained monitoring in correlation with the benchmark runs. The measurement frequency of the device reaches up to 100 Hz; hence, we are able to determine power use in high resolution. A detailed description of the measurement device can be found in [HS11].

The energy measurement device is only used for external monitoring and not connected to the master node. Internally, the DBMS is working with estimates to determine overall power consumption.

4 Experiments

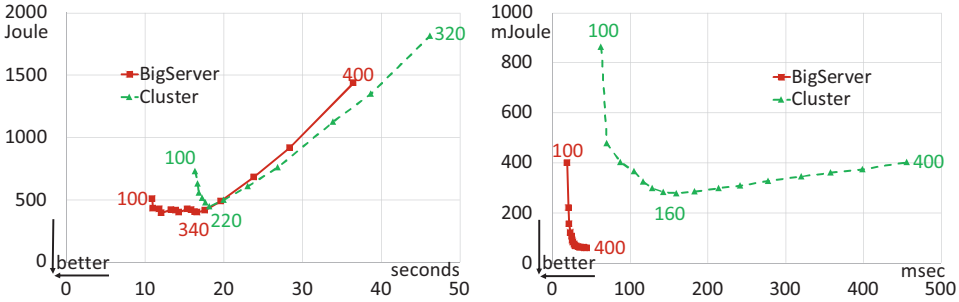
To compare the energy consumption of our cluster to that of a traditional DB server, we have processed OLAP and OLTP workloads on both platforms. First, we have run performance-centric benchmarks, i. e., DB clients continuously submit queries to the full cluster or the big server, to assess peak performance of both systems. Next, we have evaluated energy-centric benchmarks [SHK12], i. e., requests of DB clients are issued only at timed intervals thereby delaying query submission, to identify the energy-efficiency potential of the big server and the cluster. In the following, we first describe the experimental setup, before we present our results.

4.1 Experimental Setup

For all experiments, using OLTP and OLAP, we have set up the systems as previously described. A separate server, directly connected to the master node and the big server, respectively, is used as the benchmark driver, submitting queries to the cluster as well as monitoring response time and throughput. The previously introduced power measurement device is also hooked up to the benchmark driver to correlate all measurements with energy consumption.

OLAP workloads: For measuring OLAP performance and energy efficiency, we are using the well-known TPC-H benchmark with a scale factor of 300; hence, 300 GB of raw data are generated. Due to additional indexes and storage overhead, the final DB has approx. 460 GB of raw data. On the centralized server, small tables are stored on a single disk, whereas larger ones, e. g., the *LINEITEM* and *ORDERS* tables, are partitioned and distributed among all disks to increase access bandwidth and to parallelize processing on partitions.

On the cluster, the *REGION* and *NATION* tables are replicated to all nodes, while the other tables are partitioned and distributed equally among the nodes. In static benchmarks, no repartitioning occurs, even if the initial distribution leads to hotspots in the data, that impact the node's performance. If dynamic features of WattDB are enabled, the DBMS will automatically repartition as previously described.



(a) OLAP: energy/query over response time

(b) OLTP: energy/query over response time

Figure 4: Peak performance and energy consumption for both systems

OLTP workloads: For online transaction processing, we are running the TPC-C benchmark on the systems with a scale factor of 1000. Hence, a thousand warehouses were generated on the cluster, consisting of about 100 GB of data. Due to additional indexes and storage overhead, the final DB has ~200 GB of raw data at the start of the experiments.

4.2 Performance-Centric Benchmark

First, to evaluate the *peak performance* of both systems, we run performance-centric benchmarks similar to TPC-C and TPC-H on the cluster and the big server. We repeated the experiments with varying numbers of parallel DB clients (between 100 and 400) to estimate a saturation point, i. e., how many parallel queries the systems can process—without entering an overload state. Summarizing the OLAP results in Fig. 4a, the x-axis and y-axis illustrate response time in seconds resp. energy consumption per query. Numbers on the individual graphs annotate the amount of parallel DB clients for this curve progression.

Compared to the cluster, we can conclude that the big server handles queries for the same number of clients generally faster and with lower energy consumption per query, i. e., better energy efficiency. Up to 340 DB clients, the response times on the big server increase only slightly. Then, the server seems saturated and runtimes start to build up. Consequently, energy consumption per query rises.

Even for medium-sized workloads (up to 220 clients), the cluster cannot meet the energy and performance figures of the big server. Already 220 clients seem to overload the cluster, where runtimes and energy consumption for the same number of clients start to increase faster than for the big server. When stressing the cluster with more than 340 clients, WattDB crashes due to shortage of main memory.

Fig. 4b illustrates the results of the same experiments repeated using OLTP queries. Obviously, the big server is much better suited for OLTP than the cluster, as is exhibits lower query response times and also less energy consumption. Response times on the big server increase only slightly with the number of DB clients and the system does not show sat-

uration at all. Consequently, energy consumption per query improves continuously. In contrast, the cluster is saturated with 160 clients; when further increasing the number of parallel queries, response times start to increase faster.

Analyzing the access patterns of both, OLTP and OLAP queries, the different performance figures are explainable: OLAP queries read huge amounts of records, join them with (small) fact tables, and then group and aggregate the results to satisfy analytical inquiries. Hence, the reading part of these queries can run in parallel on all partitions, speeding up the query linearly with the number of disks, CPUs, and/or nodes. After having fetched the qualified records, the joins with the fact tables can also run concurrently. The final grouping and aggregation steps can be pre-processed locally for each of the parallel streams and quickly aggregated into a final result. Hence, this kind of access pattern seems to well fit both, a single, multi-core machine with lots of disks and memory but also a cluster of independent nodes, exchanging query results via network. In [SH15], we have analyzed the abilities of a cluster—with an emphasis on dynamic repartitioning of DB data—to process that kind of workloads in greater detail.

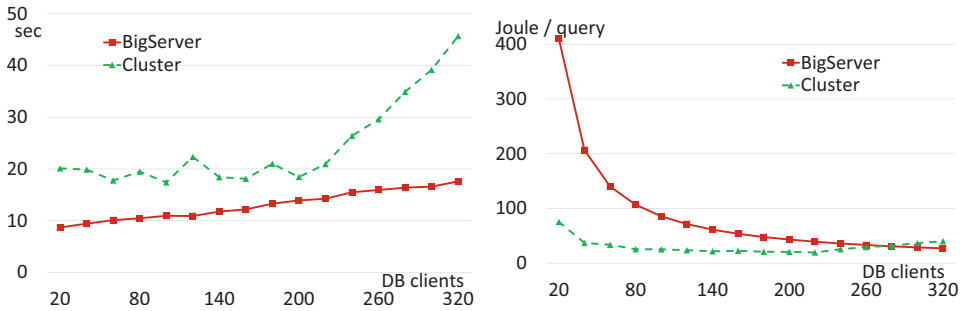
In contrast, OLTP queries touch very little data, but update records frequently. Since writers need to synchronize to avoid inconsistencies, lock information must be shared among all nodes involved. A single server with a main-memory-resident lock table can synchronize transactions much faster than a cluster, needing to exchange lock tables among nodes. Further, OLTP query operators modifying records cannot be offloaded to other nodes. Therefore, a query plan for transactional workloads is much more rigid than that of OLAP queries. In summary, it is comprehensible that a cluster of nodes is better suited for OLAP kinds of workloads than for typical transaction processing with ACID guarantees.

4.3 Energy-Centric Benchmark

After evaluating the peak performance of both configurations, we ran experiments representing average, real-world workloads. Because DB servers are often heavily underutilized, as mentioned earlier, we modified the benchmark driver to submit queries *at timed intervals* [SHK12].

Workload scaling: In each experiment, we have spawned the number of OLTP or OLAP clients between 20 and 320, sending queries to the database. Each client sends a query in a specified interval. If the query is answered within the interval, the next query is not initiated immediately, but at the start of the subsequent interval. If the query is not finished within the interval, the client waits for the answer until sending the next query. In this way, each DB client generates its share of utilization. The DBMS has to answer queries quickly enough to satisfy the DB clients, but there is no reward for even faster query evaluation. It is important to delay query submission of the clients, since we are not interested in maximizing throughput, but want to adjust the DBMS to a given workload instead, using an optimal number of nodes.[9]

[9] Otherwise, the whole benchmark would degenerate to a simple performance-centric evaluation, which is not what we intended.



(a) Query response time

(b) Energy consumption per query

Figure 5: Performance and energy consumption for varying OLAP activity levels

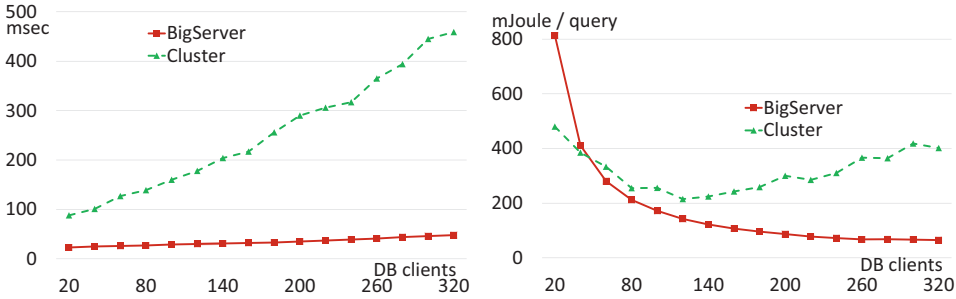
Before each workload changes, the cluster is *manually* reconfigured to best match the expected workload. We let the benchmarks run for a short warm-up time prior to measuring energy efficiency and performance to eliminate start-up cost and to identify maximum energy savings potential.

OLAP: In this experiment, we vary the number of parallel clients between 20 and 320. As before, we are using TPC-H queries on a SF300 dataset. To control utilization, the clients send queries at an interval of at least 20 seconds. Whatever comes last, the query result or the end of the interval, is the trigger for the next query.

Fig. 5 illustrates the results for the energy-centric OLAP benchmark. The left side depicts query response times of the brawny server and the wimpy cluster. As expected, the centralized machine handles queries faster than the cluster, even faster than the (pre-specified) target response time of 20 seconds per query. Therefore, the server is idle for longer time periods, still consuming energy. The cluster is meeting the target response times quite well, except for higher utilization, as observed earlier. After about 220 parallel clients, query performance starts to drop and runtimes build up.

Comparing energy consumption per query of both systems, the cluster delivers far better results for *average utilization*. Due to the cluster's scale-out and adaptation to the necessary number of nodes, its energy consumption per query stays at the same level almost the entire time, regardless of utilization. Only at high workloads, energy consumption increases because of lengthy query runtimes. The big server, with more or less static power consumption over the whole utilization spectrum, delivers bad energy efficiency for low and moderate workloads. Only at high utilization, when all the processing power of the server is needed, its energy consumption per query pushes below that of the cluster. From this experiment, we conclude that the cluster seems to be better fit for moderate OLAP workloads than the big server.

OLTP: We repeated the same experiment using OLTP queries from the TPC-C benchmark. Prior, the corresponding dataset was generated with a scale factor of 1,000. Identical to the OLAP benchmark, we scaled the DB clients between 20 and 320. Each client was waiting 40 ms between queries to simulate low and moderate workloads too.



(a) Query response time

(b) Energy consumption per query

Figure 6: Performance and energy consumption for varying OLTP activity levels

Fig. 6 plots the results of the energy-centric OLTP benchmark run. Whereas the big server exhibits query response times between 30 and 50 milliseconds, the cluster performs with processing times between 50 and 450 ms much worse. Apparently, the cluster is not well suited to process update-intensive OLTP workloads. Likewise, energy efficiency of the cluster is only better at very low workloads, below 50 parallel DB clients. While the big server consumes between ∼100 and ∼800 mJoule/query, the cluster needs between ∼200 and ∼400 mJoule/query. The big server's energy efficiency is much better at more challenging workloads, unlike OLAP workloads, where the cluster was more efficient for most scenarios. In conclusion, we have constituted a tradeoff between performance and energy consumption on the cluster. By reducing the number of nodes, both power consumption and peak performance are lowered. For moderate workloads, lower performance is tolerable and, thus, energy efficiency can be improved.

4.4 Dynamic Workloads

As previously described, the limiting factor for dynamic repartitioning is the migration cost, i. e., the performance impact and time it takes to move data between nodes. To estimate its impact on the cluster's elasticity, we have run experiments on a *dynamically adapting* cluster. Similar to the previous tests, we are running a mix of workloads against the cluster, ranging from low utilization up to heavy workloads. In this experiment, the cluster is given no warm-up times to adjust itself to a given task; instead, we are monitoring performance and energy consumption continuously.

Workloads change every 5 minutes, starting with a moderate workload of 20 DB clients, sending OLTP or OLAP queries respectively. The number of DB clients sending queries in parallel over time, i. e., the workload pattern (used in the experiments of Sect. 4.4.1), is depicted in Fig. 7.

To quantify the importance of workload forecasting, we processed the same benchmark on the cluster twice—and once on the big server for comparison. The first run on the



Figure 7: Dynamic workload over time

cluster hits the system unprepared; WattDB has to reactively adjust to the changing workload. During the second run, the DBMS was informed of upcoming workload changes (30 minutes in advance). Hence, the DBMS could proactively adjust to such a new load situation.

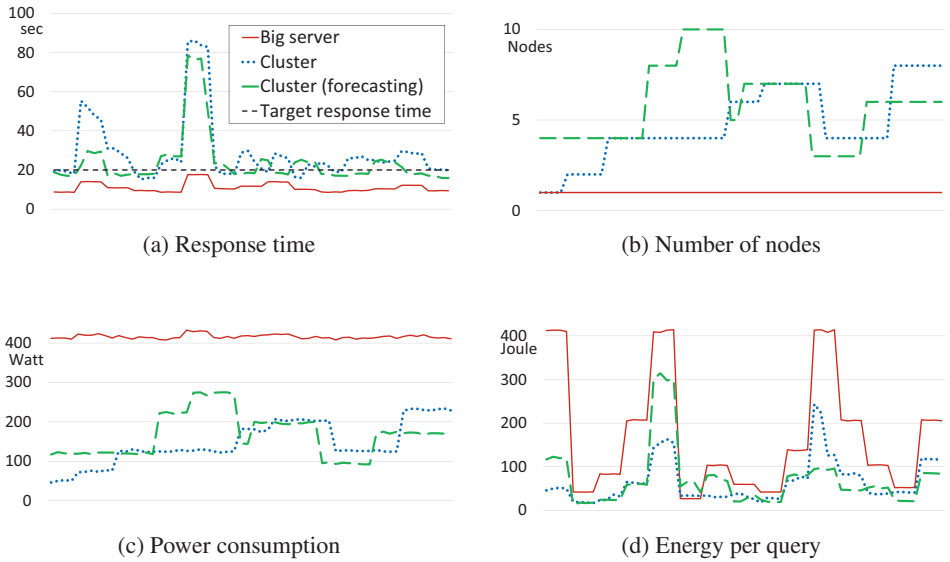
In [KHH12], we have developed a load forecasting framework to predict upcoming database workloads based on historic performance data. We observed that workloads are often repetitive and, therefore, quite easy to forecast.

4.4.1 OLAP and OLTP Processing

In the following, results of the benchmark runs are discussed separately for OLAP and OLTP. Results for the big server are visualized as *solid red* lines in Figures 8 and 9. *Dotted blue* lines represent benchmark results measured on the cluster without workload forecasting, while *dashed green* lines illustrate results of the same experiment using forecasting. The top-left plot in both figures draws the average query response time. The target response time (20 seconds for OLAP and 200 msec for OLTP drawn as *short-dashed black* lines) is included to expose the load-dependent response time deviations in the various experiments. A target response time is needed for dynamic adaption, to give the scaling algorithm a bounded optimization goal. To characterize the varying size of the cluster, the number of active nodes is depicted in the top-right chart. Underneath, the course of the overall power consumption is shown for all three experiments on the left. The resulting average energy consumption per query is plotted on the lower-right—to contrast it to the power consumption of the corresponding system.

OLAP (big server): Fig. 8 shows the results for TPC-H queries on the big server as solid red lines. The big server does not exhibit transition times between workload changes, since reconfiguration is not needed on this single-node system. Query runtimes are fast, always beating the target response time. Yet power consumption is constantly high, regardless of utilization, as already observed in earlier experiments. Average energy consumption per query is comparably high, although query runtimes are low. Because this benchmark is energy-centric, faster query runtimes do not lead to better results.

OLAP (cluster): The dotted blue lines in Fig. 8 depict the TPC-H results for the cluster without workload forecasting. The number of nodes in the cluster jitters heavily, as the system tries to adjust itself to the current workload. Reconfiguration takes time, e. g., migrating from 2 to 4 nodes requires each of the two source nodes to ship about 100 GB of data to one of the targets, hence, it takes up to 20 minutes to complete repartitioning. Therefore, query response times in this benchmark experiment are highly fluctuating and,



(a) Response time

(b) Number of nodes

(c) Power consumption

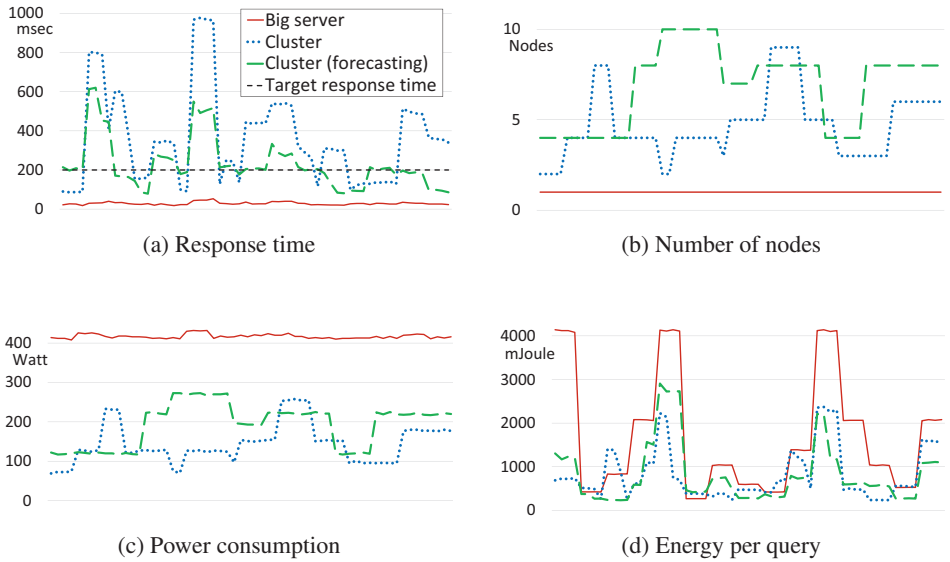
(d) Energy per query

Figure 8: Dynamic OLAP workload on the big server and the cluster

in turn, often missing the predefined deadline (target response time). Yet, as we have shown in previous experiments, the cluster, in theory, should be able to handle most of the workloads within the deadline. Due to high additional reconfiguration overhead, the nodes are overloaded. Therefore, query runtimes and avgerage energy per query remain high.

OLAP (cluster with forecasting): In this benchmark, we inform the cluster of workload changes occurring in the next 30 minutes. Hence, instead of only reacting to load changes, WattDB can now prepare for upcoming load. The dashed green lines in Fig. 8 illustrate the results. In comparison to the first run on the cluster, response times are generally lower and pass the deadline more often. Since the cluster prepares for heavy workloads in advance by scaling out to more nodes, the number of nodes is also larger in average, resulting in increased average power consumption. The energy consumption per query, however, shows a mixed picture. For low utilization, but more nodes running to prepare for upcoming events, energy consumption is higher compared to the version without forecasting. On the other hand, for higher utilization, thanks to in-advance preparations related to the forecasting approach, query runtimes are lower and exhibit better overall energy efficiency.

When comparing the big server with the cluster, we can conclude that the server is performing better, i.e., it exhibits lower query response times. On the other hand, the cluster is more energy efficient, especially during low and moderate utilization, due to its adaptation to the workload. The cluster benefits from scale-in, when performance is not needed. This translates to a steadily varying power consumption (according to the cluster size), whereas the server displays a more or less constant one. For OLAP workloads, the cluster seems an eligible alternative to a big server.



(a) Response time

(b) Number of nodes

(c) Power consumption

(d) Energy per query

Figure 9: Dynamic OLTP workload on the big server and the cluster

OLTP (big server): After running OLAP benchmarks, we have repeated the same dynamic workload with OLTP clients on the TPC-C dataset. Fig. 9 illustrates our results for experiments based on the energy-centric OLTP benchmark. As before, solid red lines characterize the behavior of the big server, which exhibits low query runtimes, but also high power consumption.

OLTP (cluster): The dotted blue lines of Fig. 9 summarize our results for the benchmark run on a cluster without load forecasting. Obviously, the response times shown are high. Because the cluster is forced to permanently repartition, response times and, in turn, energy efficiency are further worsened. Because the cluster can only react to the current workload, rebalancing starts after a sufficient change of the workload is detected. As discussed for the OLAP benchmark, this puts too much stress on the nodes and notably slows down query processing. Compared to the big server, query response times are much longer for high activity levels of the cluster. Yet, power and energy consumption are lower. Therefore, the cluster delivers better energy efficiency overall—if longer query response times are deemed acceptable.

OLTP (cluster with forecasting): The dashed green plots of Fig. 9 depict the OLTP benchmark results on the cluster using workload forecasting. Compared to the previous benchmark, the average number of nodes is higher, because WattDB is preparing for workloads in advance. As a results, query runtimes are more stable and more often pass the deadline. However, power consumption is often higher. Again, overall energy efficiency is characterized by a mixed picture: Due to preparations related to the forecasting approach, lower workloads have worse energy efficiency, but more intense workloads benefit by achieving lower energy consumption per query.

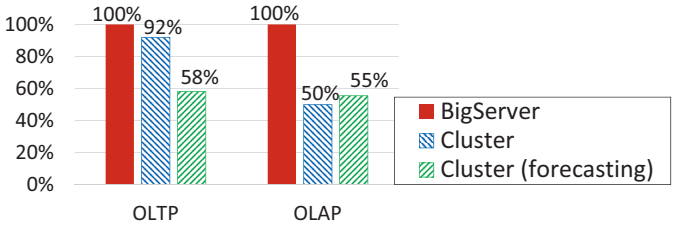


Figure 10: Energy Delay Product of OLAP & OLTP workloads

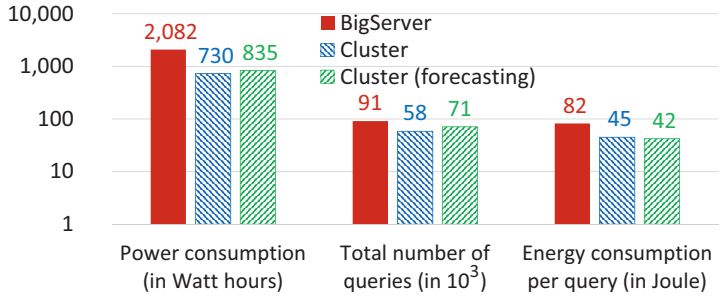
4.4.2 Energy Delay Product

The product of energy consumption (Joules) and runtime (seconds) is called *Energy Delay Product* (EDP), a metric originally stemming from chip and microprocessor design [GH96]. EDP is used as a metric combining energy consumption with performance of alternate architecture design choices. As rationale behind this metric, a system—running twice as fast and with twice the power consumption—exhibits the same EDP, hence, it is not better or worse than the original system. Another system, consuming the same amount of energy but delivering results faster will have a lower EDP. Likewise, lowering energy consumption while keeping runtimes unchanged results in a better EDP. Therefore, the Energy Delay Product is a good measure to compare the energy consumption/performance of heterogeneous systems, given the same task.

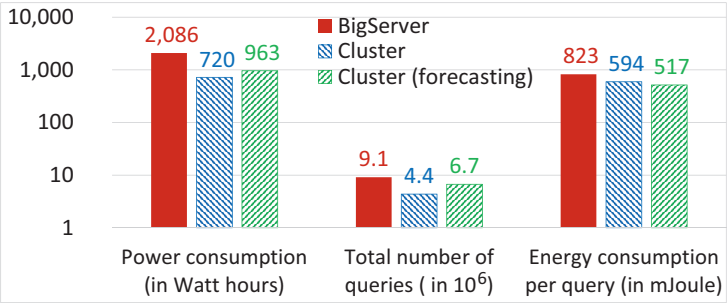
Despite different hardware platforms and power consumption figures, by calculating the EDP, we can make a square comparison of the big server and the cluster. In Fig. 10, we have summarized the EDP results from our dynamic experiments on OLTP and OLAP workloads for all three configurations. To illustrate the differences, all values are relative to the EDP of the big server. Hence, the forecasting cluster's EDP is about half of that of the big server, regardless of OLTP or OLAP workloads. For OLTP workloads, the cluster without forecasting exhibits nearly the same EDP as the big server, obviously, forecasting really payed off here. In contrast, the cluster without forecasting shows the lowest EDP under OLAP queries, even lower than the forecasting version. Apparently, preparing the cluster for upcoming workloads was a (little) waste of time and energy here. However, the cluster's EDP is lower than that of the big server; hence, with the same energy budget, we should be able to perform more work—although it might take some extra time.

4.4.3 Summary

Reviewing the results from all benchmarks, we want to extract some condensed numbers to facilitate high-level comparison and to gain a few key observations. For this reason, we have separately computed indicative numbers for the dynamic OLAP and OLTP experiments: total energy consumed (in Watt hours), total number of queries represented in units of 10^3 resp. 10^6, and average energy consumption per query in Joule resp. mJoule. These Key figures are shown in Fig. 11, where the logarithmic y-axis should be regarded.



(a) OLAP workload



(b) OLTP workload

Figure 11: Overall energy use, throughput, and avg. energy consumption per query

First, the cluster is no match for the big server considering pure performance. The centralized system does not require network communication or synchronization among nodes. Therefore, it can deliver much better query throughput than the cluster, where queries have to be distributed, results have to be collected, and the overall execution of concurrent queries on multiple nodes needs some form of synchronization to ensure ACID properties. All these factors lead to friction losses which slow down query processing.

Second, the cluster handles low and moderate workloads quite well, although the big server is still faster. Yet, the cluster requires less than half of the server's power (left-most bars in the figures). Therefore, the cluster needs less energy per query and is more energy efficient, as depicted by the right-most bars in the figures.

Third, dynamic workloads with varying utilization require preparation to adjust the number of nodes to the needs. If workloads are predictable, the cluster exhibits better energy efficiency than the single server while delivering comparable performance. Although, energy consumption of a forecasting cluster is higher, its query performance outweighs the additional wattage.

5 Conclusion

In this paper, we have examined the energy-saving potential of a clustered DBMS compared to a traditional DBMS based on a single server. An important goal of this paper was to compare performance and energy efficiency of our WattDB cluster to those of a big server. Of course, if peak DBMS performance is required during almost the entire operating time, a single-server approach has no alternative as our performance-centric benchmarks clearly reveal. However, as stated in various studies [BH09, SHK12], average utilization figures are far from continuous peak loads. A large share of database or data-intensive applications runs less than an hour close to peak utilization on workdays and is resilient w.r.t. somewhat slower response times. During the remaining time, their activity level is typically in the range of 20% – 50% and often lower. Therefore, from low- to mid-range workloads, a dynamically adjusting cluster of nodes will consume significantly less power without sacrificing too much performance. Hence, their throughput/response time requirements could be conveniently satisfied by the performance characteristics of our cluster with much less energy consumption, as confirmed by Fig. 8. Hence, the application range, where the cluster's energy efficiency largely dominates that of a single server, has quite some practical benefit.

Especially for OLAP workloads, where lots of records need to be read and aggregated without much coordination effort, a cluster seems to be a viable alternative to a single server. On the other hand, when processing OLTP workloads, where transactions need to synchronize continuously, a cluster suffers from high friction losses and is a magnitude slower than the centralized approach.

As shown, predictability of workloads and data elasticity are crucial for our approach. Fortunately, typical usage patterns are predictable and a cluster can therefore prepare for upcoming workloads. Thus, dynamically adjusting a cluster to the workload—although time-consuming—is possible.

References

[AFP+09] Michael Armbrust, Armando Fox, David A. Patterson, Nick Lanham, Beth Trushkowsky, Jesse Trutna, and Haruki Oh. SCADS: Scale-Independent Storage for Social Computing Applications. *CoRR*, abs/0909.1775, 2009.

[BFG+08] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *SIGMOD Conf.*, pages 251–264, 2008.

[BH07] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[BH09] Luiz Andre Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR Conf.*, pages 225–237, 2005.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS*, 26(2), 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[Das11] Sudipto Das. *Scalable and Elastic Transactional Data Stores for Cloud Computing Platforms*. PhD thesis, Univ. of California at Santa Barbara, 2011.

[GH96] R. Gonzalez and M. Horowitz. Energy Dissipation in General-Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277 –1284, 1996.

[Gra94] G. Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[HHOS11] Theo Härder, Volker Hudlet, Yi Ou, and Daniel Schall. Energy Efficiency is not Enough, Energy Proportionality is Needed! In *DASFAA Workshops, LNCS 6637: 226-239*, 2011.

[HS11] Volker Hudlet and Daniel Schall. Measuring Energy Consumption of a Database Cluster. In *BTW, LNI 180*, pages 734–737, 2011.

[KHH12] Christopher Kramer, Volker Höfner, and Theo Härder. Load Forcasting for Energy-Efficient Distributed DBMSs (in German). In *Proc. 42. GI-Jahrestagung 2012, LNI 208*, pages 397–411, 2012.

[LFWZ09] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling Transaction Services in the Cloud. *The Computing Research Repository*, abs/0909.1768, 2009.

[LHP+12] Willis Lang, Stavros Harizopoulos, Jignesh M. Patel, Mehul A. Shah, and Dimitris Tsirogiannis. Towards Energy-Efficient Database Cluster Design. *PVLDB*, 5(11):1684–1695, 2012.

[SBH+10] Alexander S. Szalay, Gordon C. Bell, H. Howie Huang, Andreas Terzis, and Alainna White. Low-Power Amdahl-Balanced Blades for Data-Intensive Computing. *SIGOPS Oper. Syst. Rev.*, 44(1):71–75, 2010.

[SH13a] Daniel Schall and Theo Härder. Towards an Energy-Proportional Storage System using a Cluster of Wimpy Nodes. In *BTW, LNI 214*, pages 311–325, 2013.

[SH13b] Daniel Schall and Theo Härder. Energy-Proportional Query Execution Using a Cluster of Wimpy Nodes. In *SIGMOD Workshops, DaMoN*, pages 1:1–1:6, 2013.

[SH14] Daniel Schall and Theo Härder. Approximating an Energy-Proportional DBMS by a Dynamic Cluster of Nodes. In *DASFAA Conf., LNCS 8421*, pages 297–311, 2014.

[SH15] Daniel Schall and Theo Härder. Dynamic Physiological Partitioning on a Shared-nothing Database Cluster. In *ICDE Conf.*, 2015.

[SHK12] Daniel Schall, Volker Höfner, and Manuel Kern. Towards an Enhanced Benchmark Advocating Energy-Efficient Systems. In *TPCTC Conf., LNCS 7144*, pages 31–45, 2012.

[THS10] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD Conf.*, pages 231–242, 2010.

[VCO10] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. In *VLDB Conf.*, number 1-2, pages 506–514, 2010.

Single-pass restore after a media failure

Caetano Sauer
TU Kaiserslautern
Germany
csauer@cs.uni-kl.de

Goetz Graefe
Hewlett-Packard Laboratories
Palo Alto, CA, USA
goetz.graefe@hp.com

Theo Härder
TU Kaiserslautern
Germany
haerder@cs.uni-kl.de

Abstract: When persistent storage fails, traditional media recovery first restores an old backup image followed by replaying the recovery log since the last backup operation. Restoring a backup can take hours, but log replay often takes much longer due to its random access pattern. We introduce single-pass restore, a technique in which restoration of all backups and log replay are performed in a single operation. This allows hiding log replay within the initial restore of the backup, thus substantially reducing the time and cost of media recovery and, incidentally, rendering incremental backup techniques unnecessary.

Single-pass restore is enabled by a new organization of the log archive, created by a continuous process that is easily incorporated into the traditional log archiving process. Our empirical analysis shows that the imposed overhead is negligible in comparison with the substantial benefits provided.

1 Introduction

With "big data" ever increasing in size as well as individual storage devices ever increasing in capacity, failure and recovery of storage becomes an increasing concern, at least among people with operational responsibility for large databases. Restoring a transactional database on a replacement device requires not only copying a full database backup but also replaying log records captured during the backup operation and the hours or days hence. Traditional log replay incurs many random I/O operations on the replacement device and thus often takes much longer than restoring a full backup. Thus, database software often supports and database administrators often employ incremental backups, e.g., daily backups of database pages modified since the last backup.

The incremental backup approach does not eliminate the core problem, which is the random I/O incurred on database pages during log replay. Rather, it tries to alleviate the issue by restricting the length of log replay. The problem is that incremental backups are cumbersome to maintain—both in terms of implementation effort and processing overhead. Furthermore, as transaction throughput increases due to modern hardware, log volumes are expected to grow much faster. This requires incremental backups to be taken more frequently, which not only disturbs transaction processing activity but also quickly becomes ineffective, as frequently updated pages are backed up over and over. Therefore, in case of data losses, system administrators are left with no choice but to pay the extremely high penalty of traditional media recovery.

1.1 Media recovery costs

As an example scenario for predicting the time required for media recovery, consider a database storage device of 1 TB. At 150 MB/s, a full backup or a restore operation takes about 2 hours. If 5% of all database pages change over a day, the size of a daily incremental backup is 50 GB, or 6.25 million pages of 8 KB. The restore operation for each incremental backup requires (at 1 ms average access time per page) 6,250 seconds or 1 hour and 45 minutes. Assuming that there are two log records per modified page, that the buffer hit ratio during log replay is 75%, and that the miss penalty is a random I/O operation of 4 ms, then each day of log replay takes ∼12,500 seconds or 3.5 hours. Therefore, a media failure occurring 7 days after the last full backup may take almost 16 hours to recover (2h full backup + 6 × 1.75h incremental backups + 3.5h log replay). For around-the-clock online businesses, these prospects are frightening. For some practical scenarios, this example may even be considered conservative—if data volumes and transaction throughput are much higher, recovery may easily reach the scale of multiple days.

The inefficiency of traditional log replay is emphasized in Figure 1. We measure the time required to replay a log produced by ∼7.5 million transactions, i.e., one day worth of log under a load of ∼5k transactions per minute. In a realistic scenario where the device capacity is orders of magnitude larger than the amount of main memory, single-pass restore replays all updates in 7 minutes, as opposed to almost 4 hours in the traditional setting. This experiment is described in more detail in Section 4.

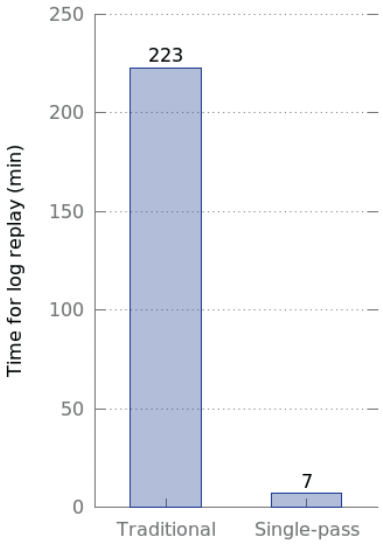


Figure 1: Time to perform log replay from a traditional vs. a sorted log

As a remedy for this situation, enterprises may choose the unconventional approach of backing up data on flash devices (or even on non-volatile RAM) to speed up recovery time, essentially decreasing the time to repair and thus improving availability. However, as observed in our previous work [SGH14], the fundamental problem is the random I/O pattern which is inherent to traditional "redo" recovery (from both system and media failures). Therefore, low-latency devices are (like incremental backups) not a solution to the problem but merely a temporary remedy—as transaction throughput increases due to new hardware and software techniques, such "fast" storage devices are expected to become the bottleneck once again. What is needed, therefore, is a software solution, preferably a simple one that requires only small incremental changes to existing techniques.

1.2 Our contribution

Single-pass restore is a novel technique that provides much faster log replay and hence media recovery. This is achieved by simply reorganizing the log into a different sort order while performing log archiving, i.e., copying the recovery log from a latency-optimized to a capacity- and cost-optimized device. With log replay practically free, incremental and differential backups lose their justification. In the example scenario described above, recovering a failed device to the most up-to-date state can take as long as simply restoring an outdated full backup, i.e., $\sim$2 instead of 16 hours.

The complexity of the new logic for log archiving and for restore operations is comparable to that of external merge sort, i.e., quite moderate. We demonstrate empirically that the overhead during normal transaction processing is negligible in a normal setting, and even in extremely fast OLTP scenarios it is as low as 1.5%.

In the remainder of this paper, Section 2 reviews related prior work, both competing approaches and technology adapted in the proposed data structures and algorithms. It also lays out some of the assumptions of our current prototype design. Section 3 introduces the new data structure for the log archive as well as algorithms for log archiving and for single-pass restore operations. Section 4 reports on our prototype implementation and its performance. Section 5 extends the basic techniques in order to overcome the simplifying assumptions. Section 6 offers a summary and some conclusions.

2 Related prior work

We divide related prior work into competing approaches, for which the new techniques may serve as alternatives, and adopted technology, upon which the proposed algorithms and data structures rely.

2.1 Competing approaches

Multiple techniques reduce the probability of a media failure or the duration of a media recovery. Among the former, hardware techniques such as disk mirroring [BG88] and RAID [PGK88] hide some failures from the operating system and all applications. Nonetheless, both mirroring and RAID can experience data loss. For those cases, recovery in the data management software remains required. Moreover, the new techniques do not require redundancy in the data store and thus save hardware costs and related costs, e.g., energy. Hot spares and replication have similar costs and dangers of data loss as mirroring and RAID. Nonetheless, mirroring is a suitable technology for the recovery log and RAID (e.g., RAID-6) is a suitable technology for the log archive.

Among the latter, incremental backups require a special in-database data structure for tracking database pages changed between backups. This data structure is expensive to

maintain during transaction processing [MN93] and very difficult to switch over in online incremental backup operations. In contrast, the new techniques hide days or weeks of log replay within the restore operation of the full backup, i.e., all effort related to incremental backups render the traditional restore techniques slower than the proposed ones. Since we consider incremental backups to be the main competing approach to single-pass restore, we provide a qualitative comparison between both approaches in Section 6.

In cases of single-page failures, traditional recovery requires media recovery. Some systems support selective media recovery, i.e., a full log scan with log replay limited to specific pages, e.g., Microsoft SQL Server [Mic14]. Some systems may also support dedicated and efficient single-page recovery [GK12] in the future. In that sense, single-page recovery is a competitor to media recovery if the recovery log embeds backward pointers not only per transaction but also per database page. However, both solutions still suffer from the random I/O pattern of traditional "redo" recovery.

Recent main-memory database systems make use of logical logging to eliminate the overhead of generating log records for every update [MWMS14]. The techniques proposed here apply to physiological or physical logging, as employed in ARIES [MHL+92], and therefore in the vast majority of database systems deployed in production businesses today. Essentially, we rely on the concept of pages as a fundamental characteristic of logging and recovery: a page is the smallest unit of fault containment and repair. Logical recovery schemes, such as those of in-memory databases, have no such concept. A comparison between logical and physiological logging techniques is beyond the scope of this paper.

2.2 Adopted technologies

The proposed techniques rely on various existing technologies. The most important among them is write-ahead logging [GR93] with physiological log records [MHL+92]. The new techniques rely on reliable and efficient access to log records, including (where single-page recovery [GK12] is used) efficient access to the history of a single database page using appropriate backward pointers among log records in the recovery log.

Log archiving frees up space on latency-optimized stable storage by copying to cost- and capacity-optimized long-term stable storage. The new techniques merely add some additional logic to log archiving, which remains a single-pass process, i.e., it reads and writes each log record only once.

Log records in a recovery log are sorted on time, in the order they are generated by executing transactions. Sorting log records differently is an old idea. For example, Gray wrote "For disk objects, log records can be sorted by cylinder, then track then sector then time" and "Since it is sorted by physical address, media recovery becomes a merge of the image dump of the object and its change accumulation tape" [Gra78]. The new techniques sort log records but not at the expense implied by Gray's design as sketched by the quoted sentences. Instead of applying sorting as a pre-processing step prior to actual data recovery, we propose maintaining a partially sorted log archive during normal operation, i.e., as part of the log archiving process.

Sorting the log is essentially an external merge sort operation, which is executed in multiple phases: run generation fed by a scan or an input process, some (zero or more) intermediate merge steps, and a single final merge step feeding directly the user of the sorted data, e.g., a merge-join algorithm. The new techniques run the traditional phases of external merge sort (input and run generation, intermediate merge steps, final merge step and production of output), but not one phase immediately after another.

2.3 Assumptions

The initial description of the proposed techniques relies on several simplifying assumptions; Section 5 alleviates most of these. Our initial assumptions include that (i) there is a single failure only; (ii) the database system and its transaction manager remain active throughout failure and recovery, i.e., throughout media loss and restore; yet (iii) the system stops transaction processing during a restore operation; and (iv) the latency-optimized stable storage for the recovery log has limited capacity such that long-term online transaction processing requires log archiving with or without the new techniques. Furthermore, we assume that (v) media recovery only requires "redo" recovery. This means that either backup operations are offline, i.e., each backup is transaction-consistent; or that uncommitted updates can be rolled back by a following crash recovery process, which feeds exclusively from the normal recovery log instead of the log archive. In practice, the latter case would be more common.

3 Partially sorted log archives

The new data structure which enables single-pass restore is a partially sorted log archive. The log archive is partially sorted because it is a collection of sorted runs instead of a single sorted file. Within each run, log records are sorted by page identifier (within the database) and by LSN. Figure 2 shows a sample recovery log for which four runs were generated in the archive. To simplify the example, we consider only three database pages, illustrated in three different shades. Due to the nature of physiological logging, only log records with page identifiers, i.e., those representing modifications, are selected for archiving. This is aligned with the assumption that the log archive supports only "redo", whereas "undo" relies on the active recovery log.

The process of generating the partially sorted log archive, including the sorting algorithm to use, is discussed in Section 3.1 below. To perform single-pass restore, all runs must be merged into a single sorted input stream. This process is discussed in Section 3.2. Finally, since failures may happen during any phase of both archiving and restore, we describe a restart mechanism that guarantees atomicity of archive and restore operations in Section 3.4.

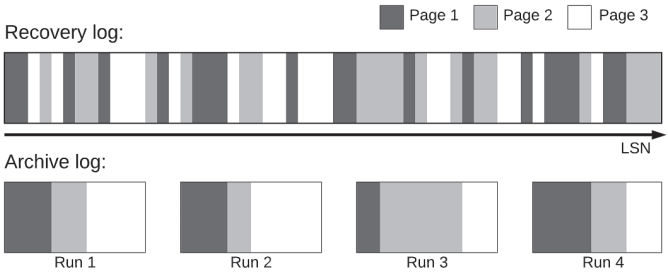


Figure 2: Visualization of partially sorted log archiving from the normal recovery log.

3.1 Log archiving logic

Traditional log archiving consists of copying log records from a latency-optimized device, where the recovery log is kept, into a capacity- and cost-optimized device. A latency-optimized device, such as a flash device or even non-volatile memory, allows more efficient transaction processing in which the log can be flushed at a higher ratio (in terms of I/O operations per second) and thus more transactions are committed per second. The log archive device, on the other hand, provides higher capacity and lower cost in terms of dollar per gigabyte. Thus, it is usually a hard disk, potentially replicated with RAID. The use of tape is discouraged in our approach, because external merge sort would be prohibitively slow.

Instead of copying "raw" log data, e.g., by copying files between file systems or physical blocks between devices, log archiving processes a log record at a time. This is done in order to compress the log archive by removing unnecessary log records such as transaction begin and commit/abort, checkpoint information, page flushing activity, etc. In essence, only log records that modify physical data, i.e., pertaining to a page, are kept. Furthermore, additional compression techniques can be applied such as removing "undo" information, combining updates into a "net change" record, or skipping aborted updates, as done in ARIES with the technique of restricted repeating of history [MP91]. Such inspection and manipulation of log records entails a certain CPU overhead to the process of log archiving. A partially sorted log archive increases the CPU effort by additionally sorting log records.

For example, in order to generate a run of 1 M log records, each log record participates in $\sim$20 comparisons (i.e., $log(n)$). In a fully tuned implementation, this amounts to 1,000 instruction cycles. Assuming the average log record size is 100 bytes, the process requires 10 CPU cycles per byte sorted. Thus, in order to maintain a sorting throughput of 200 MB/s, which is the typical sequential write speed of enterprise hard disks, 2 billion cycles per second, or 1 whole CPU core, must be reserved to the sorting procedure. This may seem like a large overhead at first, but typical OLTP workloads rarely fully utilize the CPU [TPK+13], meaning that sorting can exploit idle CPU cycles, thereby causing little to nonexistent impact on transaction processing. In Section 4, we present experiments that quantify this overhead on a real system.

The sorting algorithm used to generate runs also plays an important role. While quicksort performs quite well in terms of CPU utilization, it exhibits a periodic load-sort-store behavior, in which a chunk of data is first loaded into the sort workspace, then fully sorted, and finally written to external memory in a single pass. The disadvantage of this behavior is that it does not allow read and write operations to occur in parallel if the output device is different from the input device, which is the case in log archiving. Furthermore, I/O operations cannot overlap with the CPU effort of sorting. For these reasons, we propose the replacement-selection algorithm [Gra06], which exhibits a continuous behavior where read and write activity can occur in parallel.

One advantage of replacement selection is that it can generate runs larger than the in-memory sort workspace. This is done by holding log records in the selection tree in order to optimize the sorting distance within each run [Knu73]. However, we do not make use of this feature in our prototype implementation, because the resulting runs cannot be mapped to contiguous regions of the input, which is the recovery log. If such mapping is not provided, it becomes cumbersome to keep track of regions already archived, which is essential for resuming the log archiving process after a system crash, as we discuss in Section 3.4 below. In order to maintain this simple mapping with replacement selection, we simply eliminate the logic that places incoming log records into the current or the next run depending on their key value. Instead, log records are always assigned to a run number greater than the current root of the selection tree, regardless of their key values.

Despite influencing the behavior of the log archiving process, the choice of sorting algorithm for run generation is orthogonal to the algorithms of single-pass restore. Therefore, an implementation based on quicksort, or any other internal sorting algorithm, is equally conceivable.

3.2 Media restore logic

Restoring a database device after a media failure can occur in two different ways, depending on how the latest full backup is made available. If the backup is maintained in a disk on hot stand-by, then only log replay on this device is required. If the backup must first be fetched from a separate (potentially slower but cheaper) device, then log replay is performed on individual pages as they are restored from the backup. Regardless of which scenario is considered, the logic for log replay is the same. Therefore, we focus on the first use case: replaying the log on an existing backup copy. Later on, we discuss how log replay can be incorporated on the initial backup restore.

Log replay for single-pass restore must be executed on one page at a time in sequential order. Therefore, it requires the existing log archive runs to be merged into a single sorted input stream. Since runs are partitioned by LSN and, thus, by time, it is possible to merge only runs newer than the oldest LSN found in the backup. We omit such details in this discussion. A traditional multi-way external merge can be used to that end, and, at this point, we assume that enough memory is available to merge all runs in a single pass. Later on, we discuss techniques to alleviate this assumption.

Since log records are sorted by page identifier, the database can be restored one page at a time, by replaying all relevant updates of a page in a single read/write cycle. The end effect is that the database is restored from the backup in a single pass, thus exploiting the device sequential bandwidth for restore. A traditional log replay, on the other hand, applies log records to pages in an arbitrary order of page identifiers, which requires a substantial amount of random I/O operations. Furthermore, the efficiency of single-pass restore does not depend on the size of the buffer pool, whereas the performance of traditional log replay degrades for smaller buffer pools. These hypotheses are verified empirically in Section 4.

If the backup must first be copied from a separate capacity-optimized device such as tape or network-attached storage (which is the most common case in practice), then log replay can be executed on each page as it is copied. This corresponds essentially to a merge-join operation between the backup and the log archive, as noted in the early work of Gray [Gra78]. If the total log volume is less than the size of the database, then an up-to-date, fully operational database can be restored in the same time it takes traditional restore to simply copy an old database backup. Therefore, it is again a single-pass process, on both the log and the backup.

In some cases, the log volume to be replayed can be greater than the size of the device. This can happen on small OLTP databases with very high update frequency. In this case, the cost of single-pass restore is bounded by the size of the log, and not by the backup size. The same situation occurs in a merge-join algorithm: the cost is determined by the largest input. The point which must be emphasized is that in both situations (smaller or larger log), media recovery is performed as a single-pass, sequential operation. Note that traditional media recovery performs much worse if the log is larger. We analyze these situations empirically in Section 4.

3.3 Asynchronous merging

In order to maintain a manageable number of runs in the log archive, thereby enabling a single merge pass for log replay, an asynchronous merge daemon can be used to merge runs in the background during normal operation. The daemon can run at any specified pace and using as little memory as made available by the system. It can be scheduled to run during relatively idle periods, or to run continuously, but at a slow, unobtrusive pace. Our prototype implementation includes a simple merge daemon, which we evaluate empirically in Section 4.

If, during restore, there is not enough memory to merge the log archive in a single pass, despite proper asynchronous merging policies, a slower form of restore (but probably still much faster than traditional restore) is still possible. One obvious option is to perform offline merge steps until the number of runs decreases to the amount required by a single pass. In this procedure, the smallest runs should be merged first. Because these are probably the most recently generated runs, they will correspond to a very small percentage of the overall log archive volume, and, therefore, the added delay should be relatively small. For example, assume a scenario with 120 runs amounting to a total log archive volume of

1 TB. Despite the large overall volume, most of the runs, say, 80 of them, will probably be initial runs, i.e., runs generated directly from the sort workspace of log archiving. If the maximum merge fan-in is 100, simply merging 21 of these small runs is enough in order to perform single-pass restore. If we assume that the small runs are 1 GB in size, the offline merge incurs only a 4% delay (2% for each read and write) on the total cost of log replay. Note, however, that such scenario should be very unlikely in practice due to asynchronous merging.

3.4 Restart logic for log archiving and merging

Tolerating failures during the archiving process, for both the log and database backups, is an essential requirement. A proper implementation must not only restart the archiving process correctly, but it should also minimize the amount of work that must be repeated (e.g., copying log records multiple times). In our approach, such requirements are of utmost importance, since archiving is always active as an inherent part of normal transaction processing. This is in contrast to incremental backups, which run periodically in a fashion similar to fuzzy checkpoints [MN93].

In order to restart the log archiving and asynchronous merging daemons after a system failure, the filesystem must provide certain guarantees in order to generate runs (both initial and merged) in atomic steps. Since initial runs are generated by a continuous-output sorting algorithm—in this case replacement selection—, log records are first written into a temporary file. Once the run is finished, the file is renamed into its permanent format, which contains the LSN boundaries of the log records contained in it. This means that runs are mapped to contiguous regions of the recovery log (as discussed in Section 3.1) by means of file naming conventions. The end boundary of each file is an exclusive one, meaning that the log archive can be checked for consistency (i.e., absence of "holes") by simply comparing the end of one run with the begin of the next.

The contiguous mapping of runs to LSN ranges provides an easy and safe way of determining the "low water mark" of log archiving [GR93]. If traditional replacement selection is used instead of our adapted version, determining this mark can be quite cumbersome. Not only that, the amount of effort lost would be higher, since retaining log records in the selection tree essentially blocks the advance of the low water mark.

In order to generate initial runs atomically, file renaming must be provided as an atomic operation. Fortunately, this is the case for most common UNIX filesystems [GLI14]. Upon restart, the temporary file corresponding to the run that was being generated at the time of failure can simply be deleted, and log archiving restarts from the end boundary of the last generated run. Unfortunately, there is no simple way to avoid losing the whole temporary run, because that would require information about the in-memory selection tree, which is lost in a crash[1]. Note, however, that log archiving may be resumed while the system recovers, perhaps even in coordination with the log analysis or "redo" phases. Therefore, it is likely that the lost run will be restored by the time the system comes back up.

[1] Other sorting methods such as quicksort unavoidably lose the current run, since it is kept entirely in memory.

Asynchronous merging works in a similar way: runs are produced in a temporary file and renamed when completed. One crucial requirement is that only adjacent runs may be merged, in order to keep the contiguous mapping to LSN ranges. One additional concern in this case is that, after the file is renamed, the input runs must be deleted, which cannot be done atomically. Therefore, upon restart, the system must check all file names for overlapping LSN ranges, deleting the smaller runs it finds. This is essentially a "redo" operation that guarantees atomicity of run generation, in conjunction with the "undo" step of deleting temporary runs.

4 Experiments

4.1 Environment and prototype implementation

We implemented the basic algorithms for partially sorted log archiving and single-pass restore in Shore-MT [JPH+09], an open-source transactional storage manager which scales well on multi-core CPUs. In order to generate workload, we executed the TPC-C benchmark using the Shore-Kits package[2], which implements several standard benchmarks on top of Shore-MT. Our code is made available in an online open-source repository[3].

Being a research prototype, our system lacks some important features that would be expected on a product-level implementation. For instance, it does not support multiple devices for database pages or additional replication measures to support multiple failures. It also does not support the detection and automatic repair of media failures—single-pass restore must be invoked explicitly and while the system is offline. Furthermore, it does not implement an online backup utility—all experiments rely on backups generated explicitly also while the system is offline. However, it is important to note that these limitations are simply due to implementation effort and not conceptually inherent to the design.

The experiments described here were carried out on an Intel Xeon X5670 server with 96 GB of 1333 MHz DDR3 memory. The system provides dual 6-core CPUs with hyper-threading capabilities, which gives a total of 24 hardware thread contexts. The operating system is a 64-bit Ubuntu Linux 12.04 with Kernel version 3.11.0. Unless noted otherwise, the log archive and database backups are kept on a high-capacity hard disk.

4.2 Hypotheses under test

Our experiments are designed to test the following hypotheses:

1. In single-pass restore, the cost of log replay can be hidden in the initial phase of copying a full database backup, leading to substantially faster recovery.

[2] http://bitbucket.org/shoremt/shore-kits
[3] http://bitbucket.org/caetanosauer

2. Contrary to traditional media recovery, the time required to perform single-pass restore is independent of the amount of memory available.

3. Log archiving with run generation can be executed concurrently with high-throughput transaction processing with negligible impact on performance.

When analyzing the results of the experiments below, we refer back to these hypotheses and show that our method successfully accomplishes the goals laid out therein.

4.3 Media recovery performance

The first-glance results provided in Figure 1 demonstrate the potential gains of maintaining a partially sorted log archive. This section analyzes the performance of single-pass restore in more detail. First, we analyze the total time to perform media restore, with the goal of testing Hypothesis 1. Then, we provide a detailed analysis of log replay costs in isolation, as performed for Figure 1, but this time varying the amount of main memory available.

The restore procedure based on a sorted log archive is called "single pass" because it allows the failed device to be restored one page at a time in sequential order. The algorithm for single-pass restore is essentially a merge join between a full database backup and the log archive (i.e., the stream of log records produced by a merge of all existing runs). Since a merge join requires a single pass over each input, the device can be restored in $O(n)$ time, where n is the number of pages to be restored. In traditional restore, copying an outdated database backup alone already requires $O(n)$ I/O operations. Thus, we compare single-pass restore with copying a full backup in terms of total execution time. If Hypothesis 1 is correct, single-pass restore should be slower due to the additional merge logic, but only by a negligible margin.

In order to test this hypothesis, we performed an experiment where database files of different sizes are restored. We generate TPC-C databases of exponentially increasing scale factors, starting from 32 and up to 512, which yield sizes of 4 to 65 GB, respectively. For single-pass restore, we consider a log with about 50%–60% the size of the full backup. Note that the cost of a merge join depends only on the larger input, and so the log volume is only relevant if it is greater than the database size. As discussed in Section 3.2, even if the log is much larger than the device capacity, restore is still a single-pass operation, but, in this case, the cost is bound by the log size. Figure 3 shows the results of this experiment. Note that both axes are in logarithmic scale.

As the results confirm, there is only a marginal difference on every scale factor considered. This shows that the cost of log replay is indeed completely hidden in the process of restoring a full backup, as predicted by Hypothesis 1.

The experiment does not consider new pages which are allocated during replay of the log. Such pages can be either restored directly in the buffer pool or written to the replacement device in the same single-pass process. In the latter case, a slightly larger margin—dependent on workload characteristics—would be observed for the results above. Note that this is still much more efficient than restoring new pages during log replay, as done in

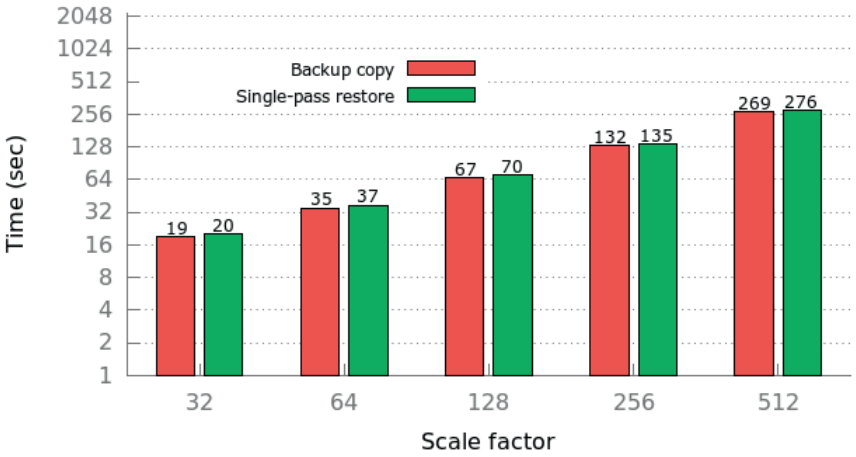


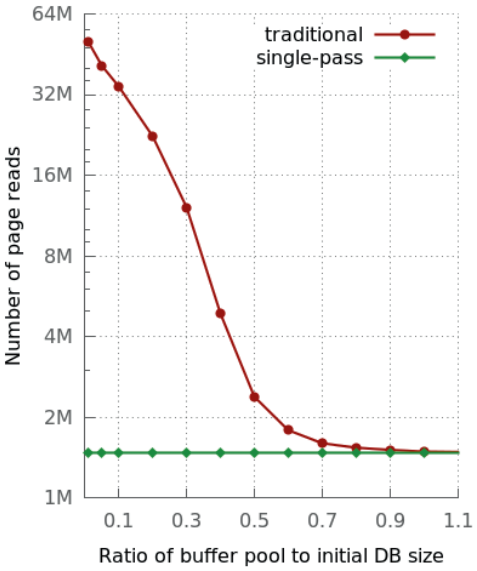
Figure 3: Time to perform single-pass restore vs. copy outdated full backup

traditional media recovery. Nevertheless, we emphasize the fact that the times measured for single-pass restore are for up-to-date recovery of device contents, whereas the baseline is just restoring an outdated backup, i.e., the long-running phase of log replay is still required afterwards. As we demonstrate on the following experiments, traditional log replay can be orders of magnitude more expensive than replay of a partially sorted log archive.

The cost for log replay in traditional restore is more complex to predict, since it depends on the amount of memory available and buffer replacement policy. If the complexity of buffer replacement is abstracted, the cost is $O(nm)$, where m is the average number of times each page gets replaced in the buffer pool. Since m grows very fast as the ratio between buffer size and device capacity decreases, so does the total cost of log replay. A sorted log archive, on the other hand, can be replayed using roughly the same number of I/O operations regardless of buffer pool size. To demonstrate these facts empirically—in support of Hypothesis 2—we compare the number of page reads observed with varying buffer pool sizes. For this experiment, we consider a log volume of 48 GB, which is generated by $\sim$7.5 million transactions, and a database with scale factor 64 (8 GB). Such discrepancy between log volume and database size is chosen deliberately in order to emphasize the costs of log replay.

Figure 4 shows the results for this experiment. As predicted by Hypothesis 2, the number of page reads required for log replay in single-pass restore remains constant. As the table on the right shows, there is actually a very small variation of up to 3 page reads in single-pass restore. This is a limitation of the Shore-MT recovery algorithms, and it can occur if the same slot on the disk is reused by pages being released and allocated multiple times. If the replay algorithm can properly detect when a page is being reformatted, such rare superfluous reads can be avoided.

Traditional log replay, on the other hand, varies drastically as the buffer size changes. For a buffer of 1% of the device size, 50 million random page reads are required, as opposed



Ratio	Traditional	Single pass
0.01	50,335,811	1,472,945
0.05	41,020,158	1,472,945
0.1	34,283,852	1,472,945
0.2	22,373,075	1,472,945
0.3	12,155,559	1,472,945
0.4	4,879,686	1,472,944
0.5	2,387,143	1,472,942
0.6	1,791,567	1,472,942
0.7	1,601,986	1,472,943
0.8	1,542,317	1,472,943
0.9	1,513,885	1,472,942
1.0	1,491,613	1,472,942

Figure 4: Number of page reads performed during log replay for varying buffer pool sizes

to 1.4 million sequential reads in the single-pass scenario. If the backup is stored in a hard disk with average read latency of 4 ms, traditional log replay would require approximately 56 hours. Single-pass restore, in this case, would be bound by the log size, which is much larger than the database size in this experiment. Even then, assuming that a read bandwidth of 150 MB/s can be sustained (i.e., typical of modern desktop hard drives), the log of 48 GB could be replayed in 3½ minutes.

4.4 Impact on transaction processing

The second set of experiments analyzes the impact of log archiving on normal transaction processing, in order to test Hypothesis 3. We compare four different scenarios with different archiving configurations. The baseline system has all log archiving features turned off, meaning that transactions can be executed at full speed without any interference. It is an impractical scenario because it does not support media recovery, but it gives a general baseline to any kind of archiving method. The second scenario represents traditional log archiving, i.e., without sorting. It requires some CPU overhead to process individual log records for suppressing log records which are irrelevant for media recovery, e.g., "undo" log records, checkpoints, transaction begin and commit/abort, etc. In the last two scenarios, partially sorted log archiving is performed, first with asynchronous merging turned off and lastly with this feature turned on.

To measure transaction processing performance in the four scenarios, we performed 30 TPC-C runs of five minutes each on a warm buffer. The environment was set up to provide

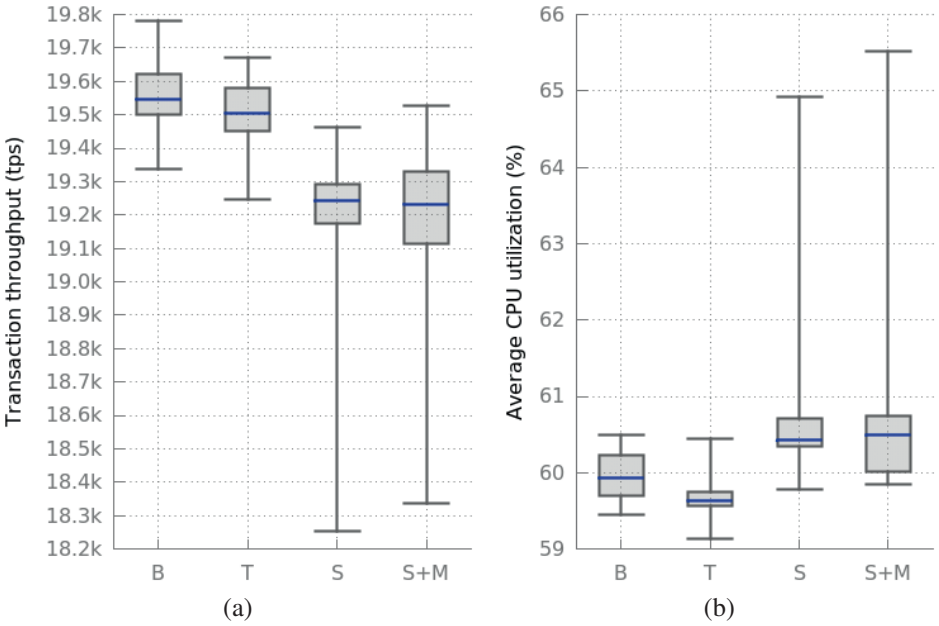


Figure 5: Transaction throughput (a) and CPU utilization (b) on 30 repetitions of each scenario

maximum transaction throughput: all CPU contexts are utilized, the complete dataset fits in the buffer pool, and the recovery log is kept in RAM. Logging to volatile memory is obviously incorrect from a transactional perspective, but it permits a worst-case analysis of the interference of log archiving, because I/O bottlenecks are eliminated and CPUs are better utilized. It also provides expectations for future non-volatile memory devices, for which write-ahead logging is a very suitable application. To provide measurements representative of current technology, we also provide results with the log on a flash device (Samsung SSD 840 Pro).

Figure 5 shows the transaction throughput (a) and the CPU utilization (b) observed in the 30 benchmark runs. Values on the y-axis represent the average transaction throughput achieved on a single 5-minute run, while the x-axis shows the four different scenarios: baseline (B), traditional archiving (T), and partially sorted archiving, both without (S) and with asynchronous merging (S+M). Note that the y-axis does not start on zero. The results are presented in a box plot format. The line in the middle of each box represents the median observation. Each box ranges from the 25-percentile on the lower boundary to the 75-percentile on the upper one. The lines extending below and above the boxes represent the minimum and maximum values observed, respectively. The box plot representation provides a summary of the statistical distribution of the observed throughput. It is therefore more useful than a simple average value.

The results show that partially sorted log archiving with asynchronous merging (S+M) has a median throughput of a little over 19.2 ktps, which is only about 1.5% less than traditional log archiving (T) at 19.5 ktps. However, it seems like partially sorted log archiving

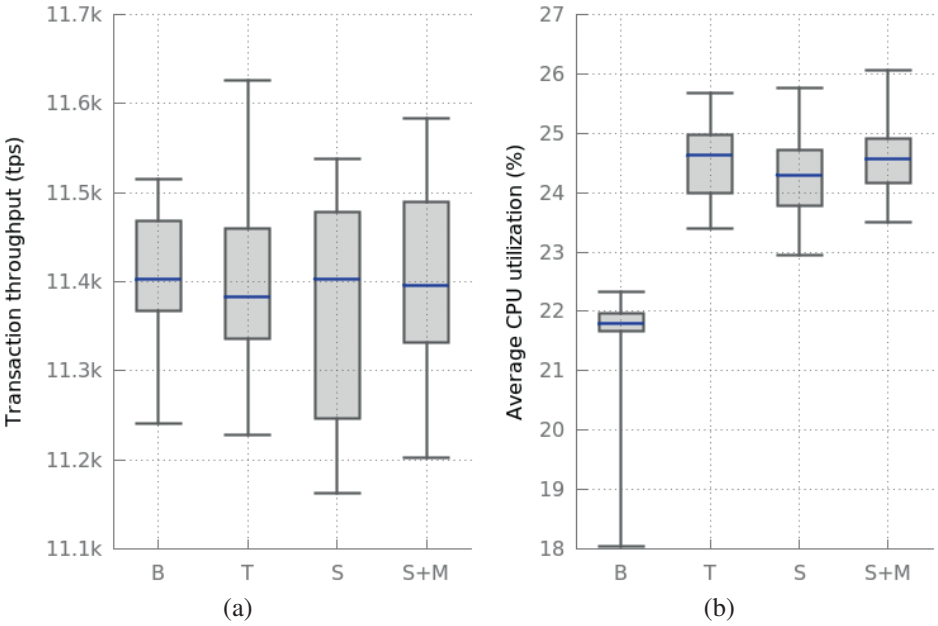


Figure 6: Transaction throughput (a) and CPU utilization (b) with log on an SSD device

produces more outliers on the lower side, as can be seen on the long bars extending below the quartile boxes. Such variation possibly originates from the randomness of page ID orders in the recovery log—less order in the input requires more swap operations in the sorting algorithm. Similar behavior can be observed on the CPU utilization results. The additional overhead of sorting translates directly into more CPU usage, again with a difference of approximately 1.5%. Note that the baseline actually has higher CPU utilization than traditional log archiving. This can be attributed to the fact that the baseline scenario has higher transaction throughput, probably because context switches are more costly than the actual computational effort in traditional log archiving.

Figure 6 presents the same analysis with the log on an SSD device. Contrary to the previous experiment, there is no observable difference between the four scenarios. This is expected because log I/O becomes the bottleneck, therefore leaving much more idle CPU time, which log archiving can exploit. This fact can be confirmed by comparing the CPU utilization of the baseline scenario with the rest, which is the only noticeable difference in the results. Note that CPU utilization fluctuates around 24%, as opposed to 60% in the previous scenario. The transaction throughput is also substantially lower.

Based on these results, we conclude that partially sorted log archiving only produces an observable impact on transaction processing performance if the I/O bottleneck is completely eliminated, which traditionally is not the case in database systems. Even if the system delivers main-memory performance for I/O, we showed that the overhead is usually as little as 1.5%, which is acceptable given the dramatical decrease on recovery time. These observations confirm Hypothesis 3 presented earlier.

Two final observations are important to conclude our analysis on transaction processing impact. First, as already hinted to above, the overhead depends on the amount of idle CPU cycles observed during the benchmark. If we consider a workload with much less logical contention, e.g., an insert-heavy benchmark such as TATP [TAT12], transactions running at full speed are able to better exploit the CPU and thus leave less room for log archiving. However, such workloads are more of a special case and not the general norm. Furthermore, an OLTP system running at full speed around the clock is an extremely uncommon scenario in real deployments. This leads to our second observation: log archiving can be scheduled to run at periods of lower transaction activity, and at an adjusted pace. Such policies are highly recommended in a product-level implementation. Nevertheless, our experiment shows that even at periods of peak activity, the overhead of log archiving can be considered negligible.

5 Extensions

The system for log archiving and media recovery as described so far is limited by the assumptions introduced above. The present section lifts some of these assumptions and introduces further promising extensions. The text here merely sketches opportunities and techniques. Future work will develop more detail on assumptions, conditions, algorithms, expected performance and scalability, tuning, etc.

5.1 Online backups

Online backups have been a standard industrial capability for a long time [Gra78]. While a backup operation is active, concurrent queries and update transactions may not only remain incomplete but may actively modify the database. Individual page images in the backup may or may not include updates by transactions concurrent to the backup operation.

After an offline backup, i.e., without concurrent transaction processing, all page images in the database backup have PageLSN values older than the start of the backup operation. After an online backup, some database pages in the back may have newer PageLSN values, but always older than the end of the backup operation. Restore from a backup taken with concurrent transaction processing must consider the PageLSN value of each database page and apply only new log records.

Thus, the log archiving logic must suspend log compression for the duration of the backup operation, i.e., combining log records. The restore logic must suppress log records preceding a page image obtained from a backup. Gray suggested to post-process each backup to obtain a transaction-consistent backup; in contrast, we propose that the required logic be in the restore operation.

5.2 Online restore

Ideally, a transactional data service remains available while one of its persistent devices or storage volumes is in media recovery. When a device first fails, if the buffer pool holds some pages of the failed device at the time of the failure, all those pages may be marked dirty immediately and do not require any further restore logic. While the database system recovers a failed device, transaction processing continues on all other devices. Incomplete transactions with updates on the failed device may, after the restore operation is complete, resume or roll back. Several of the following extensions relax this restriction.

5.3 Incremental media recovery

If the media recovery merges backup and (partitions of the) log archive in the order of database page identifiers, the replacement media can become available to transaction processing incrementally. New log records ought to capture the progress, i.e., indicate the restored and available page ranges on the replacement media. Transaction processing—including queries, updates, transaction rollback, and even restart after a crash—manages restored page ranges in the same way as media without failure or ongoing restore.

5.4 Multiple failures

During recovery from a media failure, other failures may occur. In an online restore operation, this includes transaction failures—rollback should be possible as much as forward processing. Single-page failures (other than ahead of the restore process) can proceed precisely in the same way as in the absence of concurrent media failure and restore operations.

A media failure of another device simply invokes another instance of the (single-pass) restore logic. A media failure of a replacement device simply requires another replacement device and a new invocation of the (single-pass) restore logic. A system failure (software crash) with an incomplete media recovery is the most complex case. It needs to resume the media recovery at a database page identifier definitely reached prior to the system failure. Thus, media recovery ought to log its progress occasionally as also suggested above for incremental media recovery and incremental availability.

5.5 Virtual backups and remote virtual backup

In order to obtain an up-to-date full database backup, it is not required to increase the load on a database server. Instead, merging an existing database backup and the runs in the log archive produces the same result. Such a virtual backup “creates a backup without taking a backup.”

If database transaction processing and recovery log are on one node in a network and backups, log archiving, and log archive are on a second node, then this second node can create a new, up-to-date backup without any load on the first node or the network between the nodes. A remote virtual backup has great advantages for provisioning the network, i.e., network bandwidth is required only for transaction processing but not for bursts due to across-the-network backup operations.

6 Comparison with incremental backups

In traditional systems, incremental backups are the standard technique employed to increase availability by decreasing time to repair on media failures. An incremental backup is simply the set of pages that changed since the last full or incremental backup was taken. As argued in this paper, partially sorted log archives render incremental backups completely unnecessary and thus obsolete. In this section, we provide a more in-depth discussion on the drawbacks of incremental backups as described by Mohan in the ARIES family of algorithms [MN93]. The discussion also applies to differential backups, which are taken always with respect to the last full backup.

Incremental backups require a special data structure to keep track of database pages that changed since the last backup. In order to achieve acceptable performance, access to this data structure must be done diligently, such that multiple updates to the same page in between backups only require one update on the data structure. In Mohan's method, a special transaction first collects a list of pages that must be included in the backup, resetting their state in the tracking data structure. Based on this list, the actual backup copy is performed. The LSNs generated by this transaction are used as heuristic to avoid updating the tracking state on every page update. The idea is that if the last update on a certain page (given by its PageLSN) happened after the last backup was taken, then the tracking state must have been already set by that last update. Thus, no extra access is required.

We identify two main problems with Mohan's approach. First, it adds complexity to the buffer pool logic, since the responsibility of maintaining the tracking data structure falls on normal page update operations. Our approach, in contrast, is completely independent from buffer pool operations, since log archiving is a separate process that feeds only from log data. This logical separation also implies less code complexity, less testing and maintenance effort, less concurrency control, and simpler recovery algorithms. Second, and most importantly, incremental backup techniques become ineffective if higher availability is desired. This is because frequent backups render the optimizations described by Mohan [MN93] mostly ineffective, since the ratio between updates to the tracking data structure and actual page updates increases. In contrast, partially sorted log archiving can be executed at a very aggressive pace without any logical interference on transaction processing activity. Only physical interference in the form of CPU and memory usage is expected, which may not only be an acceptable trade-off for availability, but can also be mitigated if log archiving is performed on a separate node in a network. This concern of logical separation was also raised in the original paper [MN93], which proposes a technique to

perform full backups directly from data pages on disk, i.e., without involving the buffer pool and thus not interfering with transaction processing. However, this technique is not available for incremental backups.

One argument in favor of incremental backups may be that a single copy of a frequently updated page may capture an arbitrarily large amount of updates since the last backup. Relying on the archive log for redundancy, on the other hand, means that additional space is consumed by log records of such hot-spot pages. However, as mentioned earlier, aggregating updates on the same database page into "net changes" may be easily incorporated in the processes of sorting the log (both during run generation and merging), thus reducing the log volume of hot-spot database pages.

7 Summary and conclusions

In summary, the partially sorted log archive slightly increases the cost of transaction processing, but reduces the duration of restore operations substantially. During transaction processing, log archiving partially sorts the log records, quite similar to run generation within an external merge sort. Restore operations merge a full database backup with runs in the log archive. For a small and controllable increase in system load during transaction processing, mean time to repair improves by a small factor, in some cases even an order of magnitude. Improving the mean time to repair by an order of magnitude adds another "9" to system availability, e.g., improving availability from 99.99% to 99.999%.

In conclusion, database backup and restore operations are ripe for innovation. The techniques introduced and measured above reduce the time for media recovery by a substantial factor. At the same time, with log replay practically free, there no longer is any advantage in taking differential and incremental backups. Thus, the data structures required to guide those partial backups are no longer required, simplifying database system implementation, quality assurance, and operations. Full backups no longer require access to the active database. In fact, clusters with multiple nodes may create new, up-to-date backups without access to the database server. In sum, single-pass restore based on the novel organization of the log archive may do away with all database backup operations as we know them today, in addition to speeding up restore operations and improving system availability.

Acknowledgments

We thank Pinar Tözün and Ryan Johnson for kindly and generously answering our questions about Shore-MT and Shore-Kits. Our gratitude is also extended to the Shore and Shore-MT development teams for making their code available as open source, without which this work would have been much harder to realize.

References

[BG88] Dina Bitton and Jim Gray. Disk Shadowing. In *Proc. VLDB*, pages 331–338, 1988.

[GK12] Goetz Graefe and Harumi A. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.

[GLI14] GLIBC. The GNU C Library Reference Manual. Available at: `http://www.gnu.org/software/libc/manual/html_node/Renaming-Files.html`, 2014. Accessed: 2014-10-06.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra78] Jim Gray. Notes on Data Base Operating Systems. *Lecture Notes in Computer Science*, 60:393–481, 1978.

[Gra06] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10, 2006.

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[Mic14] Microsoft. Restore and Recovery Overview (SQL Server). Available at: `http://msdn.microsoft.com/en-us/library/ms191253.aspx`, 2014. Accessed: 2014-10-09.

[MN93] C. Mohan and Inderpal Narang. An Efficient and Flexible Method for Archiving a Data Base. *SIGMOD Rec.*, 22(2):139–146, June 1993.

[MP91] C Mohan and Hamid Pirahesh. Aries-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proc. ICDE*, pages 718–727, 1991.

[MWMS14] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *Proc. ICDE*, pages 604–615, 2014.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *SIGMOD Rec.*, 17(3):109–116, June 1988.

[SGH14] Caetano Sauer, Goetz Graefe, and Theo Härder. An empirical analysis of database recovery costs. In *RDSS (SIGMOD Workshops), Snowbird, UT, USA*, 2014.

[TAT12] TATP. Telecom Application Transaction Processing Benchmark. Available at `http://tatpbenchmark.sourceforge.net/`, 2012. Accessed: Oct 2014.

[TPK+13] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proc. EDBT*, pages 17–28, 2013.

Orthogonal key-value locking

Goetz Graefe, Hideaki Kimura
Hewlett-Packard Laboratories
goetz.graefe@hp.com, hideaki.kimura@hp.com

Abstract: B-trees have been ubiquitous for decades; and over the past 20 years, record-level locking has been ubiquitous in b-tree indexes. There are multiple designs, each a different tradeoff between (i) high concurrency and a fine granularity of locking for updates, (ii) efficient coarse locks for equality and range queries, (iii) run time efficiency with the fewest possible invocations of the lock manager, and (iv) conceptual simplicity for efficient development, maintenance, and testing.

A new design introduced here is efficient and simple yet supports both fine and coarse granularities of locking. A lock request may cover (i) a gap (open interval) between two (actual) key values, (ii) a key value with its entire list of (actual and possible) row identifiers (in a non-unique secondary index), (iii) a specific pair of key value and row identifier, or (iv) a distinct key value and a fraction of all (actual and possible) row identifiers. Using specific examples such as insertions, deletions, equality queries, and phantom protection, case studies compare four prior b-tree locking techniques and the new one. Experiments show that the new technique reduces the number of lock requests yet increases transactional concurrency, improving transaction throughput for both read-only and read-write transactions. The case studies and the experiments suggest that new b-tree implementations as well as existing ones ought to adopt the new locking techniques.

1 Introduction

Many data stores support indexes on multiple attributes of stored data items. In databases, for example, these are known as secondary indexes or as non-clustered indexes. Most database management systems permit dozens of secondary indexes for each table even if fewer secondary indexes per table are more common.

Key values in secondary indexes may or may not be unique. In other words, there may be many, even thousands, of rows with the same value in an indexed attribute. In those cases, compression is useful, e.g., bitmaps instead of lists of row identifiers. Such representation choices, however, are entirely independent from transactional concurrency control, where further improvements are possible despite multiple available techniques [G 10] and decades with little progress.

1.1 Concurrency control and lock scopes

In transactional concurrency control, pessimistic techniques (locking) may focus on rows in a table across all indexes (e.g., ARIES/IM [ML 92]) or on key values in an individual index (e.g., ARIES/KVL [M 90]). When locking entries within a non-unique secondary index, the finest granularity of locking may be individual index entries (representing one row and its value in the indexed attribute) or distinct key values (such that a single lock covers all instances of the same value in the indexed attribute).

Locks on individual index entries are more desirable for insert, update, and delete operations, because they permit that two transactions modify at the same time different rows with the same value in an indexed column. The disadvantage of this approach is that queries may need to acquire a lock on each index entry, possibly thousands of entries for a single key value. Many individual lock acquisitions not only impose overhead but also incur the danger of a lock acquisition failure (e.g., deadlock) late in the process. If that occurs, the entire statement or transaction aborts and all earlier effort is wasted.

Locks on distinct key values are more desirable for search and selection operations, because a query with an equality predicate needs to retrieve (and, for serializability, to lock) all items satisfying the predicate. The disadvantage of this approach is reduced concurrency in case multiple concurrent transactions need to modify separate index entries. A single lock may cover thousands of index entries when only one is needed.

The proposed design combines these benefits, i.e., high concurrency (by using a small granularity of locking) and low overhead (by using a large granularity of locking). It is a variant of multi-granularity locking (hierarchical locking) with some new aspects that lock multiple levels within the hierarchy by a single invocation of the lock manager yet cope with a large or even infinite domain of values in the lower level of the hierarchy.

1.2 Design goals

The obvious primary design goal is correctness, e.g., in two-phase locking. The second design goal is simplicity for easier understanding, development, maintenance, and quality assurance. Thus, the proposed design is simpler than all prior ones, e.g., rendering "instant duration" locks, "insertion" lock modes, and similar creative ideas obsolete.

The third design goal is high concurrency and therefore a fine granularity of locking, wanted in particular for updates and even more particularly for updates guided by searches in other indexes. Accordingly, the proposed design enables locks on individual index entries within lists associated with distinct key values.

The final design goal is run time efficiency and thus a coarse granularity of locking, wanted in particular for large operations such as range queries and equality queries in non-unique indexes, including the read-only search required to determine a set of rows to update or delete as well as index search in index nested loops joins. All transactions benefit from a minimal number of lock manager invocations as well as the fewest and earliest lock acquisition failures in cases of contention. Accordingly, the proposed design enables locks on distinct key values and their entire lists of index entries.

1.3 Preview

The focus here is on the scope of locks in b-tree indexes and on the effect of lock scopes on the number of locks required and on the degree of concurrency enabled. The following section reviews prior techniques for record-level locking in b-tree indexes, i.e., key-value locking and key-range locking. Section 3 introduces a new design and Section 4 compares the techniques using practical example scenarios. Section 5 reports on implementation and performance. Section 6 sums up the techniques and the results.

2 Prior designs

For b-tree indexes in databases, we assume the current standard design that has data records or index entries with user contents only in the leaf nodes, also known as B^+-trees. Therefore, all designs for key-value locking and key-range locking apply to key values in the leaf nodes only.

By default, locks are possible on all key values in the index, including those marked as ghost records, also known as invalid or pseudo-deleted records. Ghost records are usually marked by a bit in the record header. For non-unique secondary indexes with a list of bookmarks attached to each distinct key value, a ghost record is a distinct key value with an empty list or one in which all remaining entries are themselves ghosts.

In this paper, the term "bookmark" means a physical record identifier (device, page, and slot number) if the table's primary data structure is a heap and a unique search key in the primary index if the table's primary data structure is a b-tree. Such primary b-tree indexes are known as a primary key index in Tandem's (now HP's) NonStop SQL, as a clustered index in Microsoft's SQL Server, and as an index-organized table in Oracle.

In most systems, an unsuccessful search in a serializable transaction locks a gap between neighboring existing key values, known as phantom protection. An alternative design inserts an additional key value (for precise phantom protection) either in the index or in some auxiliary data structure. For example, NonStop SQL includes such a key value within the lock manager (but not the data pages in the buffer pool). It seems that quality assurance and code stabilization for this design took substantial amounts of time. All methods discussed below lock the gap by locking an adjacent pre-existing key value.

2.1 ARIES/KVL "key-value locking"

ARIES/KVL [M 90] locks distinct key values, even in non-unique indexes. Each lock on a distinct key value in a secondary index covers all bookmarks associated with that key value as well as the gap (open interval) to the next lower distinct key value present in the index. A lock within a secondary index does not lock any data in the table's primary data structure or in any other secondary index.

Figure 1, copied verbatim from [M 90], enumerates the cases and conditions required for a correct implementation of ARIES/KVL. At the same time, it illustrates the complexity of the scheme. Note that IX locks are used for insertions into an existing list of bookmarks, which permits other insertions (also with IX locks) but neither queries nor deletions. In other words, ARIES/KVL is asymmetric as it supports concurrent insertions into a list of bookmarks but not concurrent deletions. Note also the use of locks with instant duration, in violation of traditional two-phase locking. This exemplifies how, from the beginning of record-level locking in b-tree indexes, there has been some creative use of lock modes that ignores the traditional theory of concurrency control but enables higher concurrency without actually permitting wrong database contents or wrong query results. Nonetheless, it substantially expands the test matrix, i.e., cost, complexity, and duration of quality assurance.

Figure 1 provides guidance for insertions and deletions but not for updates. A value change in an index key must run as deletion and insertion, but an update of a non-key field in an index record may occur in place. Non-key updates were perhaps not consid-

ered at the time; in today's systems, non-key updates may apply to columns appended to each index record using, for example, an "include" clause in order to "cover" more queries with "index-only retrieval." More importantly, toggling a record's "ghost" flag is a non-key update, i.e., logical deletion and re-insertion of an index entry. Clearly, re-insertion by toggling a previously deleted key value requires more than an IX lock; otherwise, multiple transactions might attempt, at the same time and without noticing their conflict, turning a ghost into a valid record.

		Next key value	Current key value
Fetch & fetch next			S for commit duration
Insert	Unique index	IX for instant duration	IX for commit duration if next key value *not* previously locked in S, X, or SIX mode X for commit duration if next key value previously locked in S, X, or SIX mode
	Non-unique index	IX for instant duration if *apparently* insert key value *doesn't* already exist No lock if insert key value already exists	IX for commit duration if (1) next key not locked during this call OR (2) next key locked now but next key *not* previously locked in S, X, or SIX mode X for commit duration if next key locked now and it had already been locked in S, X, or SIX mode
Delete	Unique index	X for commit duration	X for instant duration
	Non-unique index	X for commit duration if *apparently* delete key value will no longer exist No lock if value will definitely continue to exist	X for instant duration if delete key value will *not* definitely exist after the delete X for commit duration if delete key value *may* or will still exist after the delete

Figure 1. Summary of locking in ARIES/KVL.

2.2 ARIES/IM "index management"

ARIES/IM [ML 92] locks logical rows in a table, represented by records in the table's primary data structure, a heap file. Thus, its alternative name is "data only locking." A single lock covers a record in a heap file and a corresponding entry in each secondary index, plus (in each index) the gap (open interval) to the next lower key value. Compared to ARIES/KVL, this design reduces the number of locks. For example, updating a row in a table requires only a single lock, independent of the number of indexes for the table (with some special cases if the update modifies an index key, i.e., the update requires deletion and insertion of index entries with different key values).

Figure 2, copied verbatim from [ML 92], compares in size and complexity rather favorably with Figure 1, because of much fewer cases and conditions. The conditions for index-specific locking apply to the table's primary data structure. In other words, insertion and deletion always require an instant-duration lock and a commit-duration lock on either the current or the next record.

These conditions apply to secondary indexes and their unique entries (made unique, if necessary, by including row identifiers) if ARIES/IM is applied to each individual index, i.e., if "data only locking" is abandoned. The inventors claim that "ARIES/IM can be easily modified to perform index-specific locking also for slightly more concurrency

compared to data-only locking, but with extra locking costs" [ML 92] but no such implementation seems to exist. For the number of lock requests, it would combine the disadvantage of ARIES/KVL (separate locks for each index) with those of ARIES/IM (separate locks for each row in the table).

	Next key	Current key
Fetch & fetch next		S for commit duration
Insert	X for instant duration	X for commit duration if index-specific locking is used
Delete	X for commit duration	X for instant duration if index-specific locking is used

Figure 2. Summary of locking in ARIES/IM.

2.3 SQL Server key-range locking

Both ARIES/KVL and ARIES/IM reduce the number of lock manager invocations with locks covering multiple records in the database: a lock in ARIES/KVL covers an entire distinct key value and thus multiple index entries in a non-unique index; and a lock in ARIES/IM covers an entire logical row and thus multiple index entries in a table with multiple indexes. ARIES/IM with "index-specific locking" is mentioned in passing, where each lock covers a single index entry in a single index. The next design employs this granularity of locking and further introduces some distinction between a key value and the gap to its neighboring key value.

SQL Server implements Lomet's design for key-range locking [L 93]. Locks pertain to a single index, either a table's clustered index (primary index, index-organized table) or one of its non-clustered indexes (secondary indexes). Each lock covers one distinct key value (made unique, if necessary, by including the bookmark) plus the gap to the next lower key value (phantom protection by next-key locking). There is no provision for locking a distinct key value and all its instances with a single lock request. Instead, page-level locking may be specified instead of key-range locking for any clustered and non-clustered index, with additional complexity when b-tree nodes split and merge.

The set of lock modes follows [L 93]. S, U, and X locks are shared, update, and exclusive locks covering a gap and a distinct value. RS-S, RS-U, RX-S, RX-U, and RX-X locks distinguish the lock mode for the gap between keys (the mode prefixed by "R" for "Range") and for the key value itself. RI-N, RI-S, RI-U, and RI-X are "insertion" locks, all held for instant duration only and used for insertions into gaps between existing key values. The RI-_ lock modes are outside the traditional theory of concurrency control. The design lacks RS-N, RS-X, RX-N, and RU-_ modes.

SQL Server uses ghost records for deletion but not for insertion. It supports indexes on materialized views but not 'increment' locks, e.g., in "group by" views with sums and counts.

2.4 Orthogonal key-range locking

Orthogonal key-range locking is somewhat similar to key-range locking in SQL Server, but with a complete set of lock modes and completely orthogonal locks on key value and

gap. Prior-key locking (rather than next-key locking) is recommended such that a lock on a key may include a lock on the gap to the next higher key value [G 07, G 10]. Lock modes (for key value or gap) may include shared, update, exclusive, and increment.

Figure 3 illustrates combined lock modes covering key value and gap derived from traditional lock modes. Concatenation of a lock mode for key values and a lock mode for gaps defines the set of possible lock modes. Additional lock modes are easily possible, e.g., update or increment locks.

Figure 4 shows the compatibility matrix for the lock modes of Figure 3. Two locks are compatible if the two leading parts are compatible and the two trailing parts are compatible. This rule just as easily applies to additional lock modes, e.g., update or increment locks. Some compatibilities may be surprising at first, because exclusive and shared locks show as compatible. For example, XN and NS (pronounced 'key exclusive, gap free' and 'key free, gap shared') are compatible, which means that one transaction may modify non-key attributes of a key value while another transaction freezes a gap. Note that a ghost bit in a record header is a non-key attribute; thus, one transaction may mark an index entry invalid (logically deleting the index entry) while another transaction requires phantom protection for the gap (open interval) between two key values.

Gap → ↓ Key	No lock: _N	Shared: _S	Exclusive: _X
No lock: N_	N	NS	NX
Shared: S_	SN	S	SX
Exclusive: X_	XN	XS	X

Figure 3. Construction of lock modes.

Requested → ↓ Held	S	X	SN	NS	XN	NX	SX	XS
S	ok		ok	ok				
X								
SN	ok		ok	ok		ok	ok	
NS	ok		ok	ok	ok			ok
XN				ok		ok		
NX			ok		ok			
SX			ok					
XS				ok				

Figure 4. Lock compatibility.

2.5 Summary of prior designs

All prior solutions imply hard choices for the finest granularity of locking in a database index: it may be a logical row (including all its index entries – e.g., ARIES/IM) or a distinct key value with all its index entries (e.g., ARIES/KVL) or an individual index entry (requiring many locks if a distinct key value has many occurrences and thus index entries – e.g., Microsoft SQL Server). Each prior solution suffers either from limited concurrency or from excessive overhead, i.e., too many lock manager invocations.

3 Orthogonal key-value locking

What seems needed is a design that permits covering a distinct key value and all its index entries with a single lock acquisition but also, at other times, permits high concurrency among updates and transactions. The proposed design combines elements of orthogonal key-range locking (complete separation of lock modes for key value and gap), of ARIES key-value locking (a single lock for a distinct key value and all its instances), and of ARIES/IM "index-specific locking" and key-range locking (locking individual index entries). Therefore, the new name is orthogonal key-value locking.

3.1 Design goals

As stated earlier in Section 1.2, the overall design goals are correctness, simplicity, concurrency, and efficiency. More specifically, the goal is to combine the advantages of key-value locking and those of orthogonal key-range locking:

- a single lock that covers a key value and all possible instances (e.g., the list of row identifiers);
- concurrent locks on individual instances (e.g., entries in a list of row identifiers); and
- independent locks for key values and the gaps between them.

Orthogonal key-value locking satisfies the first and last of these goals and comes very close to satisfying the remaining one.

3.2 Design

The proposed new design is most easily explained in the context of a non-unique secondary index. Although representation and concurrency control are orthogonal, it might help to imagine that the representation stores, with each distinct key value, a list of bookmarks pointing to individual records in the table's primary data structure.

The proposed solution divides a set of index entries for a specific key value into a fixed number of partitions, say k partitions. Methods for lock acquisition are modified to specify not just one but multiple lock modes in a single lock manager invocation. Some designs for key-range locking, specifically in Lomet's design implemented in Microsoft SQL Server and in orthogonal key-range locking provide a precedent: in those designs, a single lock identified by a key value in an index has two modes, one for the open interval between two existing key values and one for the record with the given key value. Using two lock modes, it is possible to lock the key value without locking the gap between key values and vice versa, with some restrictions in Lomet's design.

The proposed solution extends this idea to k+1 lock modes. One of the lock modes covers the gap between two distinct key values. The other k lock modes pertain to the k partitions in the list of index entries. A lock acquisition may request the mode "no lock" for any partition or for the gap to the next key value.

Figure 5 illustrates an index record within a non-unique secondary index on an imagined employee table. Each index record contains a key value, a count of instances, and the set of instances as a sorted list of bookmarks (here, primary key values). This list is partitioned into k=4 partitions, indicated by bold and italic font choices in Figure 5. In this example, the assignment from bookmark value to partition uses a simple "modulo 4"

calculation. Index entries within the same partition (and for the same index key value) are always locked together; index entries in different partitions can be locked independently.

It makes little difference whether the lock acquisition method lists 2 lock modes (in existing designs for key-range locking) or k+1 lock modes (in the proposed design), and whether these lock modes are listed individually (e.g., in an array) or new lock modes are defined as combinations of primitive lock modes (as in Figure 3). For example, orthogonal key-range locking defines lock modes such as "XN" or "NS" (pronounced "key exclusive, gap free" and 'key free, gap shared'). They are formed from exclusive ("X"), shared ("S"), and no-lock ("N") modes on key and gap. Additional primitive lock modes, e.g., update and increment locks, can readily be integrated into orthogonal key-range locking as well as orthogonal key-value locking.

Specifically, each lock request lists k+1 modes. For only the gap between key values for an implementation with k=4 partitions per list, the requested mode might be NNNNS, i.e., no lock on any of the partitions plus a shared lock on the gap to the next key value.

A request may lock any subset of the partitions, typically either one partition or all partitions. For example, a query with an equality predicate on the non-unique index key locks all partitions with a single method invocation, i.e., all actual and possible row identifiers, by requesting a lock in mode SSSSN for k=4 partitions and no lock on the gap following the key value. An insertion or a deletion, on the other hand, locks only one partition for one key value in that index, e.g., NXNNN to lock partition 1 among k=4 partitions. In this way, multiple transactions may proceed concurrently with their insertions and deletions for the same key value, each with its own partition locked, without conflict (occasional hash conflicts are possible, however). An unusual case is a lock request for two partitions, e.g., while moving a row and thus modifying its bookmark.

Individual entries within a list are assigned to specific partitions using a hash function applied to the unique identifier of the index entry, excluding the key value. In a non-unique secondary index, the bookmarks (pointing to records in a primary data structure) serve as input into this hash function. Using locks on individual partitions, concurrent transactions may modify different entries at the same time, yet when a query requires the entire set of entries for a search key, it can lock it in a single lock manager call.

For k=1, the new design is very similar to the earlier design for orthogonal key-range locking. Recommended values are k=1 for unique indexes (but also see Section 3.4 below) and k=3 to k=100 for non-unique indexes. The optimal value of k probably depends on the desired degree of concurrency, e.g., the number of hardware and software threads.

Gender	Count	List of **EmpNo** values
'male'	89	*2*, **3**, 5, ***8***, *10*, ***12***, 13, *14*, **19**, 21, …

Figure 5. A partitioned list of bookmarks.

3.3 Lock manager implementation

Lock requests with k+1 lock modes impose a little extra complexity on the lock manager. Specifically, each request for a lock on a key value in a b-tree index must be tested in each of the k+1 components. On the other hand, in most implementations the costs of hash table lookup and of latching dominates the cost of manipulating locks.

Locks are identified (e.g., in the lock manager's hash table) by the index identifier and a distinct key value, e.g., Gender in Figure 5. This is equal to ARIES/KVL and similar to Microsoft's key-range locking and orthogonal key-range locking. Also equal is the rule that a lock request is granted if the request is compatible with all locks granted already.

The definition of lock compatibility follows the construction of Figure 4 from Figure 3: lock modes are compatible if they are compatible component by component. For example, multiple exclusive locks on the same key value but on different components or partitions are perfectly compatible. Finally, if the lock manager speeds up testing for compatibility by maintaining a "cumulative lock mode" summarizing all granted locks (for a specific index and key value), then this mode is computed and maintained component by component.

For small values of k, e.g., k=3 or even k=1 (e.g., for unique indexes), checking each component is efficient. For large values of k, e.g., k=31 or k=100, and in particular if equality queries frequently lock all partitions at once, it may be more efficient to reserve one component for the entire key value, i.e., all partitions. In a sense, this change reintroduces a lock on the distinct key value, with aspects of both ARIES/KVL (locking all actual and possible instances of a key value) and of orthogonal key-range locking (separate lock modes for key value and gap).

With the additional lock on an entire distinct key value, the number of components in the lock increases from k+1 to 1+k+1 (=k+2) lock modes for each key value in the index. This additional lock component on the entire key value permits absolute lock modes (e.g., S, X) as well as intention lock modes (e.g., IS, IX) and mixed modes (e.g., SIX). A lock request for an individual partition must include an intention lock on the entire key value. A lock request for all partitions at once needs to lock merely the entire key value in an absolute mode, with no locks on individual partitions. A mixed lock mode combines these two aspects: an SIX lock on a key value combines an S lock on all possible instances of that key value and the right to acquire X locks on individual partitions (of bookmarks associated with the key value).

Figure 6 illustrates both the lock modes on all partitions of a distinct key value and on gaps between such key values. Its left-most column lists the set of possible lock modes for the entire key value. The remainder of Figure 6 suggests lock modes formed by pairing the entire key value and the gap to the next key value. With this set of lock modes, the number of components in a lock request shrinks back from k+2 to k+1. More importantly, when a transaction needs to lock an entire key value, e.g., for a query with an equality predicate, lock acquisition and release are as fast as in traditional lock managers.

The top half of Figure 6 is equal to Figure 3; the bottom half adds intention locks for the entire key value. A lock compatibility matrix similar to Figure 4 is easily derived. In fact, derivation of these lock modes and their compatibilities is so straightforward that it is almost faster to write a program than to type them explicitly, which also simplifies adding further basic lock modes such as update and increment locks.

Another obvious optimization pertains to unique indexes locked with k=1, i.e., a single partition. In this case, it is sufficient to rely entirely on the top half of Figure 6, rendering the (one and only) per-partition lock moot.

Gap → / ↓ Entire key value	No lock _N	Shared _S	Exclusive _X
No lock: N_	N	NS	NX
Shared: S_	SN	S	SX
Exclusive: X_	XN	XS	X
Intent to share: IS_	ISN	ISS	ISX
Intent to exclude: IX_	IXN	IXS	IXX
Shared with intent to exclude: SIX_	SIXN	SIXS	SIXX

Figure 6. Construction of lock modes.

3.4 Unique secondary indexes

For unique indexes, the obvious default is to use a single partition (k=1) to match the behavior of orthogonal key-range locking. There is, however, another alternative specifically for multi-column unique keys: lock some prefix of the key like the key in a non-unique index, and lock the remaining columns (starting with the suffix of the unique key) like the bookmarks in the design for non-unique indexes.

Perhaps an example serves best to clarify this alternative. Imagine two entity types and their tables, called "student" and "course," plus a many-to-many relationship with its own table, called "enrollment." An index on enrollment may be unique on the combination of student identifier and course number, which suggests orthogonal key-value locking with a single partition (k=1).

The alternative design locks index entries as in a non-unique index on the leading field, say student identifier. In that case, queries may lock the entire enrollment information of one student with a single lock (locking all partitions in the student's list of course numbers); yet updates may lock subsets of course numbers (by locking only a single partition within a list of course numbers). With an appropriate number of partitions, the concurrency among updates is practically the same as with locks on individual course numbers.

Note that this pattern of queries is quite typical for many-to-many relationships: a precise retrieval in one of the entity types (e.g., a specific student), an index that facilitates efficient navigation to the other entity type (e.g., on enrollment, with student identifier as leading key), and retrieval of all instances related to the initial instance. The opposite example would be similarly typical – listing all students for a single course using an index on course number of enrollment – and must be well supported in a database system, its indexes, repertoire of query execution plans, etc. B-tree indexes cluster relevant index entries due to their sort order; the proposed use of key-value locking supports these queries with the minimal number of locks and of lock requests.

3.5 Summary of orthogonal key-value locking

The proposed solution combines all the advantages of locking a distinct key value and all its index entries in a single lock manager invocation, i.e., lock overhead, and most of the advantages of locking individual index entries, i.e., high concurrency. Moreover, the proposed solution opens new opportunities for concurrency control in unique secondary indexes as well as primary indexes.

4 Case studies

In order to clarify the specific behaviors of the various locking schemes, this section illustrates the required locks and the enabled concurrency of each design for all principal types of index accesses. These comparisons are qualitative in nature but nonetheless serve to highlight the differences among the schemes. Differences in performance and scalability depend on the workload; an experimental comparison follows in Section 5.

The comparisons rely on a specific (toy) example table with employee information. This table has a primary index on its primary key (unique, not null) and a non-unique secondary index on one of the non-unique columns.

Figure 7 shows some index records in the primary index. The figure shows the rows sorted and the reader should imagine them in a b-tree data structure. Note the skipped values in the sequence of EmpNo values and the duplicate values in the column FirstName. This toy example has only two duplicates but indexes on real data may have thousands. Figure 8 illustrates index records in a non-unique secondary index on FirstName. This index format pairs each distinct key value with a list of bookmarks. Unique search keys in the primary index serve as bookmarks; they are also the table's primary key here.

EmpNo	FirstName	PostalCode	Phone
1	Gary	10032	1122
3	Joe	46045	9999
5	Larry	53704	5347
6	Joe	37745	5432
9	Terry	60061	8642

Figure 7. An example database table.

FirstName	Count	EmpNos
Gary	1	1
Joe	2	3, 6
Larry	1	5
Terry	1	9

Figure 8. An example non-unique secondary index.

4.1 Empty queries – phantom protection

The first comparison focuses on searches for non-existing key values. Assuming a serializable transaction, a lock is required for phantom protection until end-of-transaction. In other words, the first comparison focuses on techniques that lock the absence of key values. The example query is "Select... where FirstName = 'Hank' ", which falls into the gap between key values Gary and Joe. Recall that we excluded in Section 2 any locking design like Tandem's in which an unsuccessful query may introduce a value into the database or the lock manager.

ARIES/KVL cannot lock the key value Hank so it locks the next higher key value, Joe. This locks all occurrences of the distinct key value without regard to EmpNo values. Thus, no other transaction can insert a new row with FirstName Hank but in addition, no other transaction can modify, insert, or delete any row with FirstName Joe.

ARIES/IM also locks the next higher key value, i.e., it locks the first occurrence of Joe and thus the row with EmpNo 3. A single lock covers the row in the table, i.e., the index entry in the primary index, the index entry in the secondary index (and any further secondary indexes), plus (in each index) the gap between those index entries and the next lower key value present in the index. While this lock is in place, no other transaction can insert a new row with FirstName Hank. In addition, no other transaction can insert new index entries (Gary, 7) or (Joe, 2), for example, because these index entries also belong into the gap locked for phantom protection, whereas a new index entry (Joe, 4) could proceed. In fact, due to the lock's scope in the primary index, no other transaction can insert any row with EmpNo 2.

Key-range locking in Microsoft SQL Server locks the first index entry following the unsuccessful search, i.e., the index entry (Joe, 3). The search in the secondary index does not acquire any locks in the primary index. Insertion of a new row with FirstName Joe is possible if the EmpNo is larger than 3, e.g., 7. Insertion of a new employee (Joe, 2) or (Gary, 7) is not possible.

Orthogonal key-range locking locks the key preceding a gap, i.e., the index entry (Gary, 1) in NS mode (pronounced 'key free, gap shared'). Insertion of new rows with FirstName Gary are prevented if the EmpNo value exceeds 1. On the other hand, non-key fields in the index entry (Gary, 1) remain unlocked and another transaction may modify those, because a lock in NS mode holds no lock on the key value itself, only on the gap (open interval) between index entries. The restriction to non-key fields is less severe than it may seem: recall from Section 2.4 that an index entry's ghost bit is a non-key field, i.e., deletion and insertion by toggling the ghost bit are possible. The lock matrix of SQL Server lacks a RangeS_N mode that would be equivalent to the NS mode in orthogonal key-range locking.

Finally, orthogonal key-value locking locks the preceding distinct key value, Gary, in a mode that protects the gap (open interval) between Gary and Joe but imposes no restrictions on those key values or their lists of EmpNo values. For example, another transaction may insert a new row with FirstName Gary or Joe and with any EmpNo value. Removal of rows with FirstName Joe has no restrictions; deletion of rows with FirstName Gary and removal of their index entries requires that the value Gary remain in the index, at least as a ghost record, until the need for phantom protection ends and the lock on key value Gary is released.

In summary, while all techniques require only a single lock manager invocation, orthogonal key-value locking provides phantom protection with the least restrictive lock scope.

4.2 Successful equality queries

The second comparison focuses on successful index search for a single key value. This case occurs both in selection queries and in index nested loops joins. The example query predicate is "…where FirstName = 'Joe' ", chosen to focus on a key value with multiple instances in the indexed column. While the example shows only two instances, real cases may have thousands. Serializability requires that other transactions must not add or remove instances satisfying this search predicate.

ARIES/KVL requires only one lock for all instances of FirstName Joe. This lock pertains to the secondary index only. Within the secondary index, the lock covers the key

value and the gap to the next lower key value, i.e., FirstName Gary. Thus, this lock also prevents insertion of a key value other than Joe, e.g., FirstName Hank.

ARIES/IM locks three rows in the table including both rows with FirstName Joe (rows 3 and 6) and the next higher key value, i.e., row 5 with FirstName Larry. The last lock is required to prevent other transactions from inserting additional instances, e.g., (Joe, 7). These locks include the gap to the next lower key in each index, i.e., both the primary index and the secondary index. Thus, they prevent insertion of new rows with FirstName Joe and EmpNo 2 or 4 as well as rows with FirstName Ken and rows with FirstName Larry and EmpNo smaller than 5.

SQL Server locks each instance of the desired key value with its unique index entry, i.e., (Joe, 3) and (Joe, 6), plus the next higher actual key value, i.e., (Larry, 5). The last lock prevents additional entries with FirstName Joe and EmpNo values greater than 6, but it also prevents insertion of additional entries with FirstName Larry and EmpNo smaller than 5 as well as key values between Joe and Larry, e.g., Ken.

Orthogonal key-range locking is similar to SQL Server locking except it locks the next lower key value from Joe instead of the next higher key value, i.e., Gary instead of Larry, and it leaves the additional record itself unlocked. A lock in NS mode (pronounced 'key free, gap shared') on key value (Gary, 1) leaves the existing index entry unlocked but it prevents insertion of new index entries with FirstName Gary and EmpNo values higher than 1, with FirstName values between Gary and Joe, e.g., Hank, and with FirstName Joe and EmpNo value smaller than 3. Only the last group is truly required to protect the result of the example query. This problem is inherent in all key-range locking schemes.

Finally, orthogonal key-value locking acquires a single lock covering all actual and possible index entries with FirstName Joe. Both adjacent key values remain unlocked, i.e., Gary and Larry. Even the gaps below and above FirstName Joe remain unlocked, i.e., other transaction are free to insert new index entries with FirstName Hank or Ken.

In summary, among all locking schemes for b-tree indexes, orthogonal key-value locking allows repeatable successful equality queries with the fewest locks and the best precision.

4.3 Other cases

A technical report compares the other principal access patterns in similar detail; the following is merely a summary.

In a range query, orthogonal key-value locking can match both open and closed intervals more precisely than all other techniques, in addition to requiring fewer locks than key-range locking and ARIES/IM in cases of duplicate key values.

In a non-key update of a single index entry, orthogonal key-range locking is the most precise method, with orthogonal key-value locking equal assuming an appropriate number of partitions.

In a single-row deletion, toggling the ghost bit in a record header is a non-key update. Thus, orthogonal key-range locking and orthogonal key-value locking are superior to the other techniques. A later system transaction performs ghost removal with latches only.

In a single-row insertion, a system transaction may first create a ghost record using latches but no locks. In the subsequent non-key update including toggling the record's ghost bit, orthogonal key-range locking and orthogonal key-value locking are superior to all earlier techniques.

The preceding cases cover all principal types of index access and compare their locking requirements. In all comparisons, orthogonal key-value locking fares very well. In queries, it is better than all prior techniques, including orthogonal key-range locking. In updates including deletion and insertion via ghost status, orthogonal key-range locking is best. Orthogonal key-value locking equals orthogonal key-range locking except in the case of hash collisions due to an insufficient number of partitions.

5 Experimental evaluation

In this section, we evaluate efficiency and concurrency of orthogonal key-value locking in order to assess the following benefits compared to prior locking techniques:

- The cost of taking orthogonal key-value locks is as low as that of taking coarse grained locks (Section 5.1).
- The concurrency of orthogonal key-value locks is as high as the concurrency with state-of-the-art prior lock modes (Section 0).
- Orthogonal key-value locking provides both of the above benefits at the same time, which no prior locking techniques could do (Section 5.3).

We implemented orthogonal key-value locking in a modified version of the Shore-MT [JPH 09] code base. In addition to orthogonal key-value locking, we applied several modern optimizations for many-core processors. Reducing other efforts and costs clarifies the differences in locking strategies and thus makes our prototype a more appropriate test bed to evaluate orthogonal key-value locking. Put differently, a system without efficient indexing, logging, etc. performs poorly no matter the locking modes and scopes. At the same time, these other optimizations amplify the need for an optimal locking technique. We found the following optimizations highly effective:

- Flush-pipelines and Consolidation Arrays [JPS 10] speed up logging.
- Read-after-write lock management [JHF 13] reduces overheads in the lock manager.
- Foster b-trees [GKK 12] make b-tree operations and latching more efficient and ensure that every node in the b-tree has always a single incoming pointer.
- Pointer swizzling [GVK 15] speeds up b-tree traversals within the buffer pool.

For the first and second optimizations, we thank the inventors for their generous guidance in applying their techniques in our code base. Our measurements confirm their observations, reproducing significant speed-ups in a different code base.

We emphasize that both orthogonal key-value locking and read-after-write lock management are modular improvements in a database code base. We first modified the existing lock modes to orthogonal key-value locking and then modified the existing lock manager in Shore-MT to read-after-write lock management.

In all experiments, we compiled our programs with gcc 4.8.2 with -O2 optimization. Our machine, an HP Z820 with 128 GB of RAM, runs Fedora 20 x86-64 on two Intel Xeon CPUs model E5-2687W v2 with 8 cores at 3.4 GHz.

To focus our measurements on efficiency and concurrency of the lock manager, we run all experiments with data sets that fit within the buffer pool. The buffer pool is pre-loaded with the entire data set (*hot-start*) before we start measuring the throughput. We also we use a RAMDisk (/dev/shm) as the logging device.

All experiments use TPC-C tables with 10 warehouses as well as workloads similar to queries within TPC-C transactions. The transaction isolation level is serializable in all experiments. Error bars in the figures show the 95% confidence intervals.

5.1 Locking overhead

The first experiment compares the overhead of taking locks using a cursor query that frequently appears in TPC-C. In short, the table schema and the query is described as follows.

```
CREATE TABLE CUSTOMER (
INTEGER WID, -- Warehouse ID
INTEGER DID, -- District ID
INTEGER CID, -- Customer ID
STRING LAST_NAME, …)
OPEN CURSOR
SELECT … FROM CUSTOMER WHERE WID=… AND DID=…
ORDER BY LASTNAME
CLOSE CURSOR
```

To process this query, virtually all TPC-C implementations build a secondary index on CUSTOMER's WID, DID, and name [TPC]. We evaluate the performance of cursor accesses to the secondary index. We compare orthogonal key-value locking that uses the non-unique key prefix (WID, DID) as its lock key to traditional granular locking that takes a lock for each index entry, i.e., ARIES/IM [ML 92] and key-range locking [L 93].

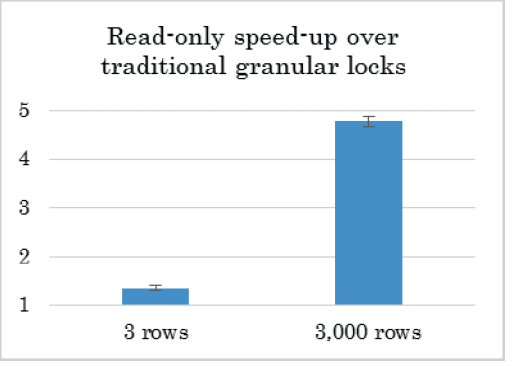


Figure 9. Lock counts and retrieval speed-up.

Figure 9 shows the speed-up achieved by orthogonal key-value locking compared to traditional granular locking in two settings. Note that the diagram shows system throughput, not just the effort spent on concurrency control.

In the first setting, we specify first-name in addition to WID and DID, touching only 3 keys per cursor access. In this case, the dominating cost is the index lookup to identify

the page that contains the records. Thus, orthogonal key-value locking reduces lock manager calls by 3× but results in only 35% speed-up.

In the second setting, we do not specify first-name, thus hitting 3,000 keys per cursor access. In this case, the dominating costs are (i) locking overhead and (ii) the cost to read each record. As orthogonal key-value locking requires only one lock request in this case, i.e., one lock covers 3,000 index entries, it effectively eliminates the first cost, resulting in 4.8× better performance.

This experiment verifies the hypothesis that orthogonal key-value locking achieves the low overhead previously achieved only by coarse-grained locking, e.g., locking entire indexes or at least pages with 100s or 1,000s of index entries.

5.2 Concurrency

The experiment above showed that the overhead of orthogonal key-value locking is as low as coarse grained locking. However, traditional coarse-grained locking is known for its low concurrency. The second experiment evaluates the concurrency enabled by orthogonal key-value locking using a write-heavy workload.

We again use the TPC-C Customer table, but this time each transaction updates the table, specifying the primary key (WID, DID, CID). Orthogonal key-value locking uses the prefix (WID, DID) as lock identifier and CID as uniquifier. We compare orthogonal key-value locking with traditional key-value locking on (WID, DID).

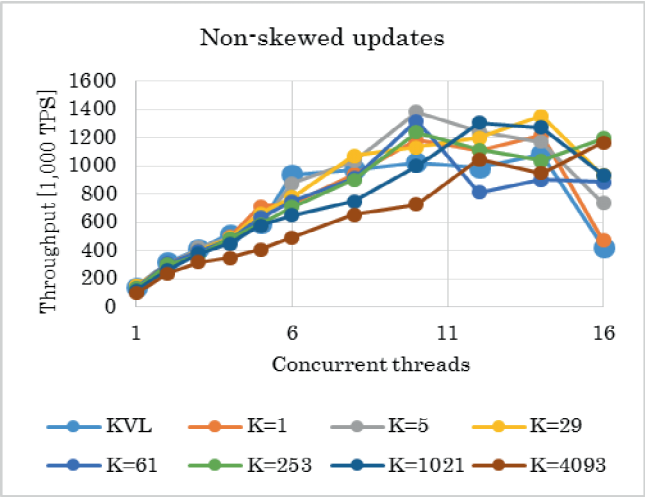


Figure 10. Transaction throughput with no skew in the requests.

Figure 10 shows the average transaction throughput, using uniformly random values for WID and DID in the retrieval requests. We varied the number of partitions in orthogonal key-value locking from k=1 (equivalent to traditional coarse-grained locking) to k=4,093. The values for k are chosen to be efficient in terms of CPU cache lines so that k+2 lock modes fit in 64 bytes or a multiply of 64 bytes.

As there is little skew, all configurations scale well with the number of concurrent threads until the system becomes over-subscribed; background threads conflict with the worker threads, e.g., logging and garbage collection in the lock manager.

One key observation is the high overhead of taking locks with extremely large values for k, e.g., k=4,093. Very large values for k cause many cache misses in the lock manager, making lock acquisition and compatibility tests expensive.

Figure 11 shows the result of the same experiment with skewed values of WID and DID in the updates; 80% of the transactions choose the first WID and 80% choose the first DID. Thus, concurrent transactions attempt to access the same index entries.

In this case, traditional key-value locking as well as orthogonal key-value locking with extremely few partitions (e.g., k=1) suffer from logical lock contention, hitting a performance plateau with only 3 threads. With larger k, orthogonal key-value locking is more concurrent and enables higher transaction throughput, except for an extremely large value (k=4,093), which causes cache-misses and physical contention in the lock queue.

This experiment verifies that, when there are many threads that are concurrently accessing the same data within the database, orthogonal key-value locking with a reasonably number of partitions achieves high concurrency because the lock compatibility of orthogonal key-value locking with many partitions is effectively equivalent to a very fine granularity of locking.

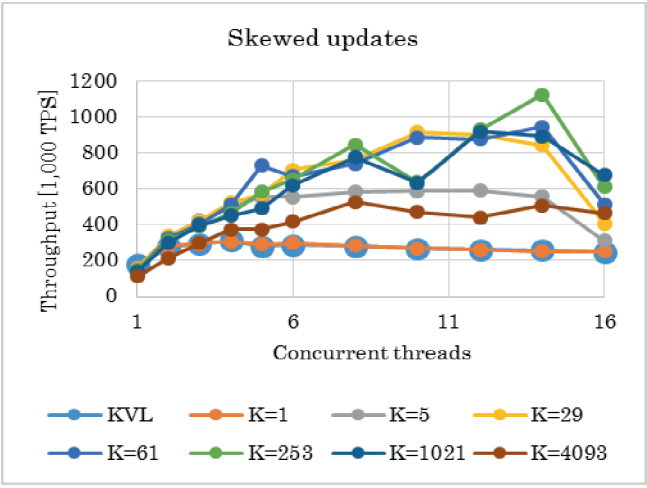


Figure 11. Transaction throughput with skew in the requests.

5.3 Mixed workloads

The experiments above show cases in which orthogonal key-value locking is as good as traditional locking modes, either coarse-grained or fine-grained. In this experiment, we show a more realistic case; a read-write mixed workload. In such a workload, both low overhead and high concurrency are required at the same time.

The workload consists of three transaction types; SELECT, INSERT, and DELETE. The mixture ratio is 40% SELECT, 40% INSERT, and 20% DELETE. Each transaction

chooses WID with skew (90% of them chooses the first WID). All transactions use the STOCK table in TPC-C as shown below:

```
CREATE TABLE STOCK(
INTEGER WID, -- Warehouse ID
INTEGER IID, -- Item ID
…)
```

A transaction uniformly picks an item id and selects, inserts, or deletes tuples. STOCK table may or may not have the particular pair of WID and IID. Thus, SELECT and DELETE might hit a non-existing key. We compare four different lock modes to handle these cases:

- ARIES/KVL [M 90] locks unique values of the user-defined index key including a neighboring gap. Our implementation uses a lock on (WID) in this experiment.
- Key-range locking (KRL) [L 93] somewhat separates lock modes for unique index entries and gaps between their key values. As it lacks some lock modes (e.g., 'key free, gap shared'), our implementation takes a lock in a more conservative lock mode (e.g., 'key shared, gap shared', known as RangeS_S) in such cases.
- Orthogonal key-range locking (OKRL) [G 07, G 10, KGK 12] uses orthogonal lock modes for unique index entries and gaps between them.
- Orthogonal key-value locking (OKVL) uses (WID) as the lock key with (IID) as uniquifier. Orthogonal key-value locking acquires only one lock per transaction, but the lock contains partition modes for the accessed IIDs.

The next experiment uses 14 threads and 253 partitions in orthogonal key-value locking.

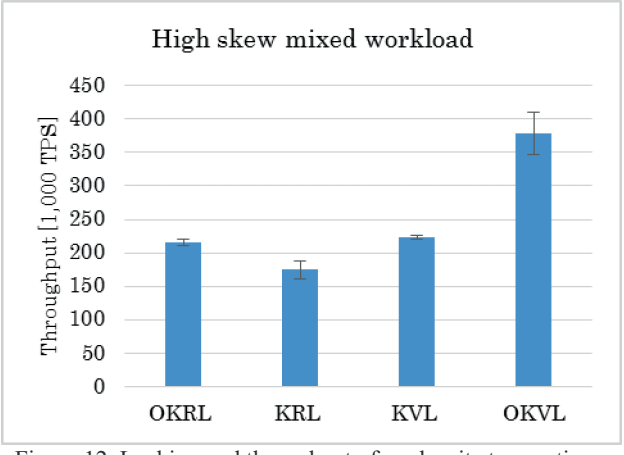


Figure 12. Locking and throughput of read-write transactions.

Figure 12 shows that with a highly skewed workload, ARIES/KVL (coarse-grained lock) fails to enable concurrency among updates. Its degree of parallelism is very low here, even if other experiments (Figure 9) show that is very efficient in single-threaded execution because it takes only one lock per transaction. On the other hand, key-range locking and orthogonal key-range locking enable more concurrent transactions but acquire many locks, which makes these techniques suffer from lock waits and even deadlocks.

In contrast, orthogonal key-value locking achieves both low overhead and high concurrency at the same time. It takes only one lock per transaction yet allows transactions to run concurrently, for 1.7-2.1× better transaction throughput. Thus, this experiment verifies our hypothesis that only orthogonal key-value locking combines the benefit of a coarse and a fine granularity of locking in b-tree indexes.

6 Conclusions

In summary, the new technique called orthogonal key-value locking combines design elements as well as advantages of (i) key-value locking in ARIES (a single lock for an equality query), of (ii) key-range locking (locking individual index entries for high concurrency), and of (iii) orthogonal key-range locking (independent lock modes for gaps between key values). The principal new techniques are (i) partitioning (only for the purpose of locking) each set of bookmarks within a non-unique secondary index and (ii) specifying individual lock modes for each partition and for the gap (open interval) between distinct key values.

A detailed case study compares the locking methods for ordered indexes (in particular b-trees) and demonstrates that orthogonal key-value locking is superior to all prior techniques for queries. For updates, it effectively equals the best prior method, which is orthogonal key-range locking. It performs sub-optimally only if ghost records must not be used for some reason or if the number of partitions is chosen so small that hash collisions within a list of bookmarks lead to false sharing and thus to lock conflicts.

Our prototype validates the anticipated simplicity of implementation in any database management system that already uses similar, traditional techniques, namely key-range locking or key-value locking. Like orthogonal key-range locking, and unlike prior lock techniques for b-tree indexes, orthogonal key-value locking permits automatic derivation of combined lock modes (e.g., for entire key value and gap) and automatic derivation of the lock compatibility matrix. It seems possible to automate even the derivation of test cases including expected test outcomes.

An experimental evaluation validates the insights gained from the case study: in situations with high contention, orthogonal key-value locking combines the principal advantages of key-value locking and (orthogonal) key-range locking. A read-only experiment shows retrieval throughput increase by 4.8 times and a mixed read-write workload shows a transaction throughput 1.7-2.1 times better than the prior techniques. Thus, we expect to find the new techniques in new implementations of b-trees and hope that they will replace the locking schemes in existing implementations. We hope that this study will affect design and implementation of databases, key-value stores, and (modern, b-tree-based) file systems alike.

Acknowledgments: Wey Guy, Ryan Johnson, Ilia Petrov, and the reviewers gave excellent feedback on earlier versions of this paper.

[BHG 87] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman: Concurrency control and recovery in database systems. Addison-Wesley 1987.

[BS 77] Rudolf Bayer, Mario Schkolnick: Concurrency of operations on b-trees. Acta Inf. 9: 1-21 (1977).

[CAB 81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, Robert A. Yost: A history and evaluation of System R. Comm. ACM 24(10): 632-646 (1981).

[G 78] Jim Gray: Notes on data base operating systems. Advanced course: operating systems. Springer 1978: 393-481.

[G 07] Goetz Graefe: Hierarchical locking in b-tree indexes. BTW 2007: 18-42.

[G 10] Goetz Graefe: A survey of b-tree locking techniques. ACM TODS 35(3) (2010).

[GKK 12] Goetz Graefe, Hideaki Kimura, Harumi Kuno: Foster b-trees. ACM TODS 37(3) (2012).

[GMB 81] Jim Gray, Paul R. McJones, Mike W. Blasgen, Bruce G. Lindsay, Raymond A. Lorie, Thomas G. Price, Gianfranco R. Putzolu, Irving L. Traiger: The recovery manager of the System R database manager. ACM Comput. Surv. 13(2): 223-243 (1981).

[GVK 15] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, Alistair Veitch: In-memory performance for big data. Accepted for publication, VLDB 2015.

[JHF 13] Hyungsoo Jung, Hyuck Han, Alan David Fekete, Gernot Heiser, Heon Young Yeom: A scalable lock manager for multicores. ACM SIGMOD 2013: 73-84.

[JPH 09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35.

[JPS 10] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, Anastasia Ailamaki: Aether: a scalable approach to logging. PVLDB 3(1): 681-692 (2010).

[KGK 12] Hideaki Kimura, Goetz Graefe, Harumi A. Kuno: Efficient locking techniques for databases on modern hardware. ADMS@VLDB 2012: 1-12.

[L 93] David B. Lomet: Key range locking strategies for improved concurrency. VLDB 1993: 655-664.

[LY 81] Philip L. Lehman, S. Bing Yao: Efficient locking for concurrent operations on b-trees. ACM TODS 6(4): 650-670 (1981).

[M 90] C. Mohan: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. VLDB 1990: 392-405.

[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1): 94-162 (1992).

[ML 92] C. Mohan, Frank E. Levine: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD 1992: 371-380.

[TPC] http://www.tpc.org/tpcc/results/tpcc_perf_results.asp.

Query Processing

Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse

Dirk Habich, Patrick Damme, Wolfgang Lehner

Technische Universität Dresden
Institut für Systemarchitektur
Professur für Datenbanken
Nöthnitzer Strasse 46
01187 Dresden
dirk.habich@tu-dresden.de
patrick.damme@mailbox.tu-dresden.de
wolfgang.lehner@tu-dresden.de

Abstract: In Hauptspeicher-zentrischen Architekturansätzen für Datenbanksysteme müssen für die Anfrageverarbeitung sowohl die Basisrelationen als auch die Zwischenergebnisse im Hauptspeicher gehalten werden. Des Weiteren ist der Aufwand, um Zwischenergebnisse zu generieren, äquivalent zum Aufwand, um Änderungen an den Basisrelationen durchzuführen. Daher sollten zur effizienten Anfrageverarbeitung die Zwischenergebnisse so organisiert werden, dass eine effiziente Verarbeitung im Anfrageplan möglich ist. Für diesen Bereich schlagen wir den durchgängigen Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse vor und haben das Ziel, eine ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse zu entwickeln. Dieser Artikel gibt einen Überblick über unser Konzept und die wissenschaftlichen Herausforderungen. Darüber hinaus führen wir erste Ansätze zur Optimierung von leichtgewichtigen Kompressions- und Transformationsverfahren ein, die aus Sicht der Anfrageverarbeitung eventuell sinnvoll sind, um einen effizienten Wechsel des Kompressionsverfahrens durchzuführen.

1 Einführung

Die Bedeutung von In-Memory-Datenbanksystemen steigt zunehmend sowohl im wissenschaftlichen als auch im kommerziellen Kontext. In-Memory-Datenbanksysteme verfolgen einen Hauptspeicher-zentrischen Architekturansatz, der sich dadurch auszeichnet, dass alle performancekritischen Operationen und internen Datenstrukturen für den Zugriff der Hauptspeicherhierarchie (z.B. effiziente Nutzung der Cachehierarchie etc.) ausgelegt sind. Üblicherweise gehen In-Memory-Datenbanksysteme davon aus, dass alle relevanten Datenbestände auch vollständig in den Hauptspeicher eines Rechners oder eines Rechnerverbundes (Cluster-Konfiguration) abgelegt werden können.

Im Gegensatz dazu stehen (klassische) Festplatten-zentrische Architekturansätze, deren originäres Ziel es ist, den Zugriff auf den Externspeicher zu optimieren; effiziente Primär-

und Sekundärindexstrukturen werden gepflegt und der Anfrageoptimierer entscheidet für eine gegebene Anfrage, wie und in welcher Reihenfolge auf die auf Externspeicher abgelegten Daten zugegriffen wird. Die Optimierung der internen Datenrepräsentation ist in diesen Architekturansätzen nachgelagert, da Externspeicherzugriffe um Größenordnungen langsamer als direkte Zugriffe auf den Hauptspeicher sind. Mit der Entwicklung hin zu Hauptspeicher-zentrischen Architekturansätzen ändert sich dieser Aspekt jedoch komplett: Jeder Zugriff auf ein Zwischenergebnis ist genauso teuer wie ein Zugriff auf die Basisdaten; entsprechend ist auch der Aufwand, Zwischenergebnisse zu generieren (bis auf den zusätzlichen Aufwand der Protokollierung) ähnlich teuer wie Änderungen an Basisrelationen.

Auf Grundlage des veränderten Architekturverständnisses bieten sich zwei orthogonale Optimierungstechniken für die effiziente Behandlung der Zwischenergebnisse an. Auf der einen Seite sollten Zwischenergebnisse nicht mehr zum Beispiel durch entsprechend angepasste Code-Generierung [Neu11] oder durch den Einsatz kooperativer Operatoren [KSHL13] produziert werden. Auf der anderen Seite sollten Zwischenergebnisse (wenn sie beispielsweise nicht vermeidbar sind) so organisiert werden, dass eine effiziente Weiterverarbeitung ermöglicht wird. In diesem Kontext greifen wir uns den optimierten Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse in Hauptspeicherzentrischen Datenbankarchitekturen heraus und haben zum Ziel, eine *ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* zu entwickeln. Mit der expliziten Kompression aller Zwischenergebnisse soll (i) die Effizienz einzelner Datenbankanfragen bzw. der Durchsatz einer Menge an Datenbankanfragen erhöht werden, da der Hauptspeicherbedarf für Zwischenergebnisse reduziert und der Mehraufwand zur Generierung der komprimierten Form möglichst gering gehalten wird und (ii) die durchgängige Betrachtung der Kompression von den Basisdaten bis hin zur Anfrageverarbeitung etabliert wird. Diese Art der Anfrageoptimierung wurde bereits diskutiert [CGK01], jedoch nicht näher untersucht, da der Rechenaufwand für die Komprimierung den Nutzen der Transferkostenreduktion bisher überstieg. Durch die immer größer werdende Schere zwischen Rechenleistung und Speicherbandbreite bei modernen Mehrkernprozessoren [QRR+08] und der Weiterentwicklung insbesondere leichtgewichtiger Kompressionsverfahren [SGL10, WPB+09, ZHNB06] verliert dieses Argument jedoch zusehends seine Gültigkeit. Dennoch ist es gerade wegen der zusätzlichen Kosten von Kompressionsverfahren wichtig, dass eine *Balance* zwischen den reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten hergestellt wird, um die Gesamtausführungszeiten einer Anfrage zu minimieren.

Zur Erreichung unseres Ziels liefert dieser Artikel einen ersten aber nicht umfassenden Beitrag, wobei folgende Schwerpunkte betrachtet werden:

1. Einen ausführlichen Überblick über den Stand der Technik leichtgewichtiger Kompressionsverfahren und deren Einsatz in Datenbanksystemen (Abschnitt 2).
2. Eine strukturierte Formulierung unseres Ziels und der notwendigen wissenschaftlichen Herausforderungen, denen wir uns zur Zielerreichung stellen müssen (Abschnitt 3).
3. Aufbauend auf der Zielformulierung präsentieren wir den aktuellen Forschungs-

stand. Neben der effizienten Umsetzung von leichtgewichtigen Kompressionsverfahren haben wir uns mit einer neuen Klasse von Transformationsverfahren beschäftigt, um Daten von einem Kompressionsschema in ein anderes Schema effizient zu überführen. Derartige Transformationen sind im Rahmen einer Anfrageverarbeitung sinnvoll, da sich die Datencharakteristiken während der Verarbeitung in der Regel stark ändern können und damit Einfluss auf die Anwendbarkeit mit Blick auf Effizienz und Kompressionsrate der Kompressionsverfahren haben (Abschnitt 4).

4. Im Abschnitt 5 evaluieren wir unsere leichtgewichtigen Kompressions- und Transformationsverfahren und zeigen das Potential für eine Integration in die Anfrageverarbeitung auf.

2 Stand der Technik

Alle Arbeiten zu Kompression in relationalen Datenbanksystemen können in zwei große Bereiche eingeteilt werden. Der erste Bereich beschäftigt sich mit der Kompression von Relationen auf verschiedenen Granularitätsstufen: (1) Zeilenebene [GRS98, IW94, RHS95], (2) Spaltenebene [AMF06, MF04] und (3) Seiten- bzw. Blockebene [PP03]. Kompression auf Spalten- und Zeilenebene ermöglicht es unter bestimmten Voraussetzungen, Datenbankoperationen direkt auf den komprimierten Daten auszuführen, während komprimierte Seiten vor ihrer Benutzung vollständig entpackt werden müssen. Der zweite Bereich befasst sich mit der Kompression von Indexstrukturen, die oftmals durch ihre Anzahl in einem relationalen Datenbanksystem (pro Relation mehrere Indizes) ähnlich viel Speicherplatz wie die Relationen selbst belegen und daher auch großes Kompressionspotential bieten. Bezüglich der Granularität lässt sich die Kompression von Indexstrukturen auf der Ebene von Seiten [BB07, GRS98], Schlüsseln [BU77, BLM+09, Com79, SDF+00] und Tupel-IDs (TIDs) [BLM+09, Tro03] einteilen. TID-basierte Indexstrukturen mit vielen TIDs pro Schlüssel sind ein Spezialfall schlüsselbasierter Indexstrukturen und besitzen oft ein höheres Kompressionspotential.

Der hier betrachtete Aspekt der Kompression von Zwischenergebnissen während der Anfrageverarbeitung wurde bisher kaum oder nur unvollständig betrachtet, da üblicherweise der Zugriff auf Externspeicher die Gesamtausführungskosten dominiert. Dabei ist dieser Ansatz unabhängig davon, ob ein Festplatten- oder Hauptspeicher-zentrischer Architekturansatz für Datenbanksysteme betrachtet wird, da Zwischenergebnisse (1) möglichst immer im Hauptspeicher gehalten werden sollten und (2) großes Kompressionspotential bieten. Bisher werden jedoch Zwischenergebnisse nur insofern komprimiert, als dass versucht wird, solange wie möglich Datenbankoperationen auf komprimiert vorliegenden Basisdaten durchzuführen [AMF06, Goy83, GS91, RS06]. Jedoch erlauben nicht alle Datenbankoperatoren die Verarbeitung komprimierter Daten, so dass die Datensätze beziehungsweise ganze Seiten während der Anfrageverarbeitung im Anfrageplan einmalig dekomprimiert und ab dieser Stelle weiter unkomprimiert verarbeitet werden müssen. Abbildung 1 fasst nochmals die beschriebenen Ansatzpunkte für Kompression in Datenbanken zusammen und zeigt die Einordnung unserer aktuellen Forschung in diesem Bereich.

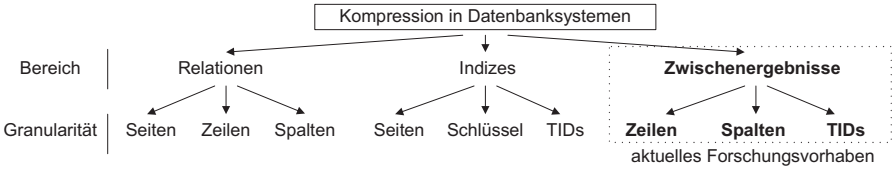


Abbildung 1: Kompression in relationalen Datenbanksystemen.

2.1 Arten von Kompressionsverfahren

Im Forschungsbereich der klassischen Kompression existiert eine Vielzahl an wissenschaftlichen Publikationen. Klassische Kompressionsalgorithmen, wie z.B. arithmetisches Kodieren [WNC87], Huffman [Huf52], Predictive Coding [CIW84] und Lempel-Ziv (LZW) [LZ76, ZL77], erzielen hohe Kompressionsraten, sind jedoch rechenintensiv und werden deshalb oft als *schwergewichtige* Kompressionsverfahren bezeichnet. Aufgrund der Rechenintensität werden die durch Kompression verringerten I/O-Transferzeiten wieder aufgehoben, so dass klassische Verfahren für die Kompression von Zwischenergebnissen im Hauptspeicher höchstwahrscheinlich nicht geeignet sind.

Speziell für den Einsatz in Datenbanksystemen wurden *leichtgewichtige* Kompressionsalgorithmen entwickelt, die verglichen mit klassischen Verfahren aufgrund der Einbeziehung von Kontextwissen ähnlich hohe und teilweise auch höhere Kompressionsraten erzielen, aber sowohl eine viel schnellere Kompression als auch Dekompression erlauben.

Beispiele für leichtgewichtige Kompressionsalgorithmen sind unter anderem Domain Kodierung (DC) [WPB+09], Wörterbuch-basierte Kompression (Dict) [ALM96, BMK99, LC86], reihenfolgeerhaltende Kodierungen [ALM96, ZIL93], Lauflängenkodierung (RLE) [Bas85, RH93], Frame-of-Reference (FOR) [GRS98, ZHNB06] und verschiedene Arten von Nullwertkomprimierung [AMF06, Reg81, RH93, WKHM00]. Abbildung 2 zeigt eine Klassifikation einer Auswahl relevanter leichtgewichtiger und schwergewichtiger Kompressionsalgorithmen.

Für die verschiedenen Datentypen eines Datenbanksystems werden unterschiedliche Kompressionsverfahren eingesetzt, die sich grob in Verfahren zur Kompression von (1) numerischen Werten und (2) Zeichenketten einteilen lassen. Für numerische Werte wie Ganzzahlen, Gleitkommazahlen oder Datumsangaben wurde eine Vielzahl von Kompressionsverfahren [CGK00, GRS98, Reg81, RH93] entwickelt. Da für die Kompression von Zeichenketten, wie z.B. CHAR, VARCHAR, BLOB, weitere Kontextinformationen zur Verfügung stehen, wurden für diese Datentypen speziell angepasste Verfahren [BHF09, BCE77, CGK01] entworfen.

Neben diesen Datentyp-spezifischen Kompressionsverfahren, existiert ebenfalls eine aktuelle Arbeit, die sich der Kompression von Verbundresultaten widmet [MLL13]. In diesem Ansatz wird ein Wörterbuch-basiertes Kompressionsverfahren vorgestellt, welches sowohl Informationen aus dem Anfrageplan als auch dem Datenbankschema nutzt, um Verbundergebnisse sehr kompakt zu repräsentieren. Ziel dieser Arbeit war, die Ergebnisübertragung in einem Client-Server-Umfeld zu optimieren.

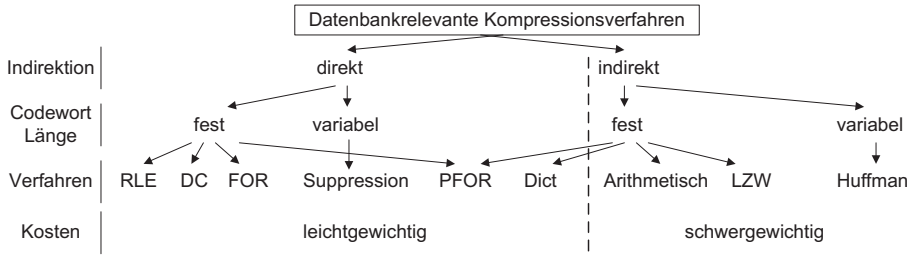


Abbildung 2: Klassifikation verschiedener Kompressionsverfahren.

2.2 Auswahl von Kompressionsverfahren

Ein wichtiger Aspekt bei der Kompression in Datenbanksystemen ist die Auswahl des besten Verfahrens für die vorliegenden Daten. Die Auswahl des Kompressionsverfahrens bzgl. der Datencharakteristiken von Relationen wird in unterschiedlichen Arbeiten adressiert. Dabei lassen sich diese Arbeiten nach ihrer Granularität klassifizieren. Zum einen existieren verschiedene Arbeiten [GRS98, GS91, WKHM00], die auf Ebene einzelner Attribute die Auswahl vornehmen, so dass für jeden Attributtyp einer Relation ein angepasstes Kompressionsverfahren genutzt wird. Ein Spezialfall bzw. eine Vereinfachung bildet dabei die Auswahl eines Kompressionsverfahrens für komplette Spalten [AMF06, MF04] in einem spaltenorientierten Datenbanksystem. Zum anderen existieren Arbeiten [HRSD07, RS06], die Kompressionsverfahren auf Ebene der Datensätze für Relationen auswählen. Die Auswahl ist dabei nicht auf einzelne Kompressionsverfahren beschränkt, sondern kann sich auch auf Kombinationen verschiedener Kompressionsverfahren [LLR06, RS06], wie beispielsweise die Kombination von Lauflängenkodierung und Domain-Kodierung, erstrecken. Als Grundlage für die Auswahl eines Kompressionsverfahrens [AMF06, CGK01, HRSD07] werden klassischerweise Angaben wie die Anzahl der Ausprägungen, die durchschnittliche Anzahl an Wiederholungen, die Lokalität der Daten und der geschätzte Kompressionsmehraufwand herangezogen.

2.3 Integration von Kompression in Datenbanksysteme

Komprimierte Relationen werden in klassischen, zeilenbasierten und Festplatten-zentrischen Datenbanksystemen genau wie unkomprimierte Relationen in Seiten auf dem externen Speicher organisiert. Hierbei unterscheidet man die Ablage in unabhängig komprimierte Seiten [GRS98, PP03], bei denen alle nötigen Informationen zur Dekompression (z.B. ein Wörterbuch) innerhalb einer Seite vorhanden sind, Relationen-komprimierte Seiten [MF04] und Relationenübergreifend-komprimierte Seiten [LLR06] (z.B. mit globalen Wörterbüchern). Aufgrund der sehr hohen Kosten für den Datentransfer zwischen Externspeicher und Hauptspeicher sind auch komplexere bzw. schwergewichtige Kompressionsverfahren (wie z.B. Lempel-Ziv [AMF06] oder Huffman [HRSD07, RS06, RSQ$^+$08]) für die Kompression der auf dem Externspeicher abgelegten Daten sinnvoll.

Wie bereits angesprochen, werden Kompressionstechniken auch auf Ebene der Datenbank-Indexstrukturen eingesetzt. Hierbei werden zwei Klassen unterschieden: Zum einen existieren Indexstrukturen, die Kompression inhärent besitzen, wie z.B. Bitmap-Indizes [Joh99], bei denen ein Schlüssel je nach Anzahl der Ausprägungen durch wenige Bits ersetzt wird. Als weiteres Beispiel gelten blockbasierte B+-Bäume [Sto88], bei denen nur ein Schlüssel für eine Menge von Sätzen genutzt wird. Zum anderen existieren Indexstrukturen, bei denen Kompression explizit genutzt wird. Beispiele hierfür sind B-Bäume [BU77, Com79, GRS98, HCL+90], B+-Bäume [BU77, BLM+09] und Präfixbäume [BHF09] mit komprimierten Schlüsseln sowie komprimierte Bitmap-Indizes [Joh99, MZ92, WOS06].

Für die Anfrageausführung existieren zur Zeit drei verschiedene Strategien, die sich in Zeitpunkt und Art der Datendekompression unterscheiden und sich auf den klassischen Festplatten-zentrischen Architekturansatz beziehen. Bei *eager decompression* [IW94] werden die Daten vom externen Speicher dekomprimiert, sobald diese in die Seiten des DB-Puffermanagers geladen werden. Der DB-Puffermanager hält deshalb nur Seiten mit dekomprimierten Daten. Der Hauptvorteil dieser Strategie ist, dass die Anfrageverarbeitung (bzw. das Verarbeitungsmodell) nicht angepasst werden muss, da nur auf dekomprimierten Daten gearbeitet wird. Bei *lazy decompression* [AMF06, GRS98, Goy83, GS91, HRSD07, RS06, RHS95, WKHM00, ZHNB06] wird die Eigenschaft ausgenutzt, dass bestimmte Operationen wie beispielsweise Projektion, Selektion und Gleichverbund direkt auf komprimierten Daten durchgeführt werden können und somit die Dekompression der Daten möglichst spät bzw. erst bei Bedarf im Anfrageplan durchgeführt wird. Bei der dritten Strategie, *transient decompression* [CGK01], werden die Daten vor der Ausführung der relationalen Operatoren, die nicht auf komprimierten Daten arbeiten können, dekomprimiert; als Ausgabe der Operatoren wird jedoch die komprimierte Repräsentation genutzt, die temporär für jeden Datensatz vorgehalten wird.

3 Vision: Ausgewogene Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse

Unser zentrales Ziel bzw. unsere Vision ist die Untersuchung und Realisierung einer *durchgängigen* Kompression von Zwischenergebnissen zur effizienten Anfrageverarbeitung. Ohne Beschränkung der Allgemeinheit konzentrieren wir uns auf den Bereich der Hauptspeicher-zentrischen Datenbanksysteme, um eine umfassende Behandlung der Kompression in derartigen System zu etablieren. Die gewonnenen Erkenntnisse können bzw. sollen konzeptionell auf die klassischen Festplatten-zentrischen Datenbanksysteme übertragen werden.

In [CGK01] ist die Optimierung der Anfrageverarbeitung mittels komprimierter Zwischenergebnisse bereits diskutiert worden, jedoch nicht näher untersucht. Durch die immer größer werdende Schere zwischen Rechenleistung und Speicherbandbreite bei modernen Mehrkernprozessoren [QRR+08] und der Weiterentwicklung insbesondere leichtgewichtiger Kompressionsverfahren [SGL10, WPB+09, ZHNB06] verliert dieses Argument je-

doch zusehends seine Gültigkeit. Dennoch ist es gerade wegen der zusätzlichen Kosten wichtig, dass eine *Balance* zwischen den reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten hergestellt wird, um die Gesamtausführungszeiten einer Anfrage zu minimieren und damit ergibt sich unser Ziel der ausgewogenen Anfrageverarbeitung. Um dieses Gleichgewicht zu erzielen, muss nicht nur die Anfrageverarbeitung sondern auch der dazu notwendige Teil der Anfrageoptimierung adressiert werden. Ziel ist es, die Kompression von Zwischenergebnissen als parametrierbaren Bestandteil in die Anfrageoptimierung zu integrieren.

Unsere Betrachtung erstreckt sich dabei von der Untersuchung und Optimierung leichtgewichtiger Kompressionsansätze (struktureller Aspekt) über die Möglichkeit, Datenbankoperatoren direkt und soweit wie möglich auf einer komprimierten Datenform auszuführen (operationaler Aspekt) bis hin zur Integration in die Optimierungskomponente (Optimierungsaspekt), um situationsabhängig entscheiden zu können, wann und welche Verfahren für eine gegebene Datenbankanfrage eingesetzt werden sollen. Diese drei Aspekte stellen im Wesentlichen auch die wissenschaftlichen Herausforderungen dar, denen wir uns zur Zielerreichung stellen müssen. Nachfolgend sind die drei Herausforderungen etwas näher erläutert.

3.1 Struktureller Aspekt

Wie bereits in Abschnitt 2 dargelegt ist, existiert eine Vielzahl von leichtgewichtigen Kompressionsverfahren, so dass wir kein neues Verfahren entwickeln, sondern uns um effiziente Implementierungen kümmern wollen. Insbesondere wollen wir erörtern, wie die einzelnen Kompressionsverfahren parallelisiert werden können und somit von den verschiedenen Arten von Parallelität moderner Prozessoren (Pipeling, SIMD, Many-Core) profitieren. Die effiziente Umsetzung der Algorithmen ist ein wichtiger Schwerpunkt, um eine Balance zwischen reduzierten Transferzeiten und den durch Kompression erhöhten Verarbeitungszeiten erzielen zu können.

Neben der Anwendung klassischer leichtgewichtiger Kompressionsverfahren beziehungsweise einer Kombination von Kompressionsverfahren ist die Transformation von komprimierten Daten ohne vorherige vollständige Dekompression im Bereich der Anfrageverarbeitung von enormer Bedeutung. Dieser Bereich ist bisher ungenügend bis kaum bearbeitet worden. Aus Sicht der Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse ist es oft sinnvoll, für die Eingabedaten und Ausgabedaten eines einzelnen Operators unterschiedliche Kompressionsverfahren einzusetzen oder zumindest unterschiedliche Parameter eines Kompressionsverfahrens zu nutzen. So ist bei einem Selektionsoperator häufig das Kompressionspotential für die Ausgabedaten höher als für die Eingabedaten, da die Anzahl der Ausprägungen in einer Spalte oder mehreren Spalten reduziert wird. Wird nun zum Beispiel Domainkodierung mit 24-bit Codewörtern für die Eingabedaten eingesetzt, dann können 16-bit Codewörter für die Ausgabedaten ausreichend sein. Die Transformation von 24-bit zu 16-bit Codewörtern sollte dabei ohne die aufwändige und vollständige Dekompression durchgeführt werden. Dies könnte zum Beispiel erreicht werden, indem Techniken (Permutation von Bytes mittels SIMD Instruktionen) aus [SGL10] eingesetzt

werden. Ähnliche Techniken auf Basis neuer Instruktionen (z.B. PEXT) in modernen Prozessoren (wie z.B. Haswell) erlauben zudem die Transformation von Codewörtern, die nicht an Byte-Grenzen (byte aligned) ausgerichtet sind oder auch Transformationen zwischen unterschiedlichen Kompressionsverfahren wie z.B. Nullwertkomprimierung und Frame-of-Reference.

3.2 Operationaler Aspekt

Neben dem bisher beschriebenen strukturellen Aspekt, ist der operationale Aspekt für eine ausgewogene Anfrageverarbeitung ebenfalls ein zentraler Baustein, da physische Planoperatoren entworfen und umgesetzt werden müssen, die als Eingabe komprimierte Daten entgegennehmen und als Ergebnis ebenfalls wiederum komprimierte Daten bereitstellen, so dass möglichst alle Planoperatoren mit komprimierten Zwischenergebnissen arbeiten können. Die Herausforderung bei dieser Aufgabe besteht darin, dafür zu sorgen, dass der Aufwand für die Integration verschiedener Kombinationen von Kompressionsverfahren möglichst gering ist. Das bedeutet, dass die Anzahl der Planoperatoren linear mit der Anzahl der integrierten Kompressionstechniken skalieren soll.

Wie in Abschnitt 2 gezeigt worden ist, existieren drei unterschiedliche Integrationsmöglichkeiten von Kompressionstechniken in Planoperatoren, die wir ebenfalls betrachten werden. Insbesondere die Integrationsvariante *transient decompression* [CGK01], bei der Daten partiell und temporär vor der Ausführung der Operationen, die nicht auf komprimierten Daten arbeiten können, dekomprimiert werden, wobei als Ausgabe der Operation jedoch die komprimierte Repräsentation genutzt wird, ist für unsere *ausgewogene Anfrageverarbeitung* enorm wichtig. Dieses Verfahren bildet die Basis für unseren Ansatz, wobei der grundlegende Unterschied jedoch darin besteht, dass Zwischenergebnisse in verschiedene Repräsentationen unterschiedlicher Kompressionsverfahren im Anfrageplan überführt werden und somit wiederholt komprimiert als auch dekomprimiert werden müssen. Dies erlaubt den Wechsel des *optimalen* Kompressionsverfahrens im Ausführungsplan in Abhängigkeit der Operatoren und der Dateneigenschaften. So kann zum Beispiel ein selektiver Operator die Anzahl der Ausprägungen einer Relation so reduzieren, dass ein Kompressionsverfahren, das für die Basisdaten ungeeignet war, für die Kompression der Zwischenergebnisse nach dem Operator gut geeignet ist. Diese Art der Anfrageausführung wurde bisher ausschließlich für die Auswertung boolescher Ausdrücke auf komprimierten Bitmap-Indizes [AYJ00] genutzt. Wir wollen es für beliebige Attributdatentypen für Zwischenergebnisse der Anfrageverarbeitung mit beliebigen Operatoren unterstützen.

Die wiederholte Kompression von Zwischenergebnissen bzw. des Operatorzustands existiert bisher nur für einzelne Datenbankoperationen, die auf externen Speicher angewiesen sind. Ein Beispiel hierfür ist externes Sortieren variabel langer Zeichenketten [YZ03, YZ07]. Dabei werden durch Kompression des Operatorzustandes (sortierte Teilbereiche) die Transferkosten innerhalb der Speicherhierarchie gesenkt und somit die Verarbeitungszeit reduziert. Allerdings beschränken sich die bisherigen Arbeiten auf diesem Gebiet nur auf die Reduktion der Transferkosten zum externen Speicher, es wird nicht

auf die Besonderheiten Hauptspeicher-zentrischer Architekturen z.B. mit Direktzugriff auf bestimmte Datenbestände geachtet; insbesondere beschränken sich die verwandten Arbeiten nur auf das Sortieren variabel langer Zeichenketten.

3.3 Optimierungsaspekt

Nach Betrachtung des strukturellen und des operationalen Aspektes kann die eigentliche Anfrageoptimierung betrachtet werden. Auf dieser Ebene müssen sowohl reduzierte Transferkosten als auch der Mehraufwand für Kompression und Dekompression bei der Suche eines optimalen Ausführungsplanes mit einbezogen werden. Hierfür existieren bereits Erweiterungen für den Anfrageoptimierer [AYJ00, CGK01], die bei der Suche nach einem optimalen Plan die Entscheidung berücksichtigen, wann und wo dekomprimiert werden soll. Auch die Wahl des Kompressionsverfahrens für Zwischenergebnisse und die Auswahl von Operatoralternativen, die auf komprimierten Daten arbeiten können, sind für die Anfrageoptimierung [AYJ00] wichtige Einflussgrößen. Bisher wurden diese Optimierungen ausschließlich für die Auswertung boolescher Ausdrücke auf komprimierten Bitmap-Indizes [AYJ00] durchgeführt. Unser Ziel ist, ein allgemeines Verarbeitungsmodell zu entwerfen, bei dem Kompression in die Anfrageverarbeitung und Anfrageoptimierung mit einbezogen wird. Deshalb sind weitere Optimierungstechniken für die komprimierungssensitive Anfrageoptimierung zu entwickeln, welche einen großen Einfluss auf Verarbeitungszeiten von Anfragen haben können. Unsere Anfrageoptimierung soll auf Basis eines Kostenmodells erfolgen, wobei dieses Kostenmodell explizites Wissen über die leichtgewichtige Kompression und Transformation hat. Dieses Wissen soll über eine empirische Evaluierung der Verfahren erlangt werden.

4 Kompressions- und Transformationsverfahren

Wie im vorherigen Abschnitt beschrieben, ist für unser Ziel der *ausgewogenen Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* der strukturelle Aspekt ein extrem wichtiger Baustein, der am Anfang betrachtet werden muss. Zur umfassenden Bearbeitung dieses Aspektes bauen wir ein Repository mit leichtgewichtigen Kompressions- und Transformationsverfahren auf, wobei Abbildung 3 einen Überblick über unseren aktuellen Stand zeigt. Wir haben uns dabei zunächst auf drei bedeutende Klassen von Kompressionsverfahren konzentriert:

1. *Run Length Encoding (RLE)* mit den Algorithmen *RleSeq* und *RleSimd*
2. *Dictionary Encoding (Dict)* mit den Algorithmen *SimpleDictSeq* und *AnalyzingDictSeq*, wobei AnalyzingDict als OrderPreservingDict bezeichnet werden kann
3. *NullSuppression (NS)* mit den Algorithmen *4-Wise Null Suppression* und *4-Gamma Coding* [SGL10].

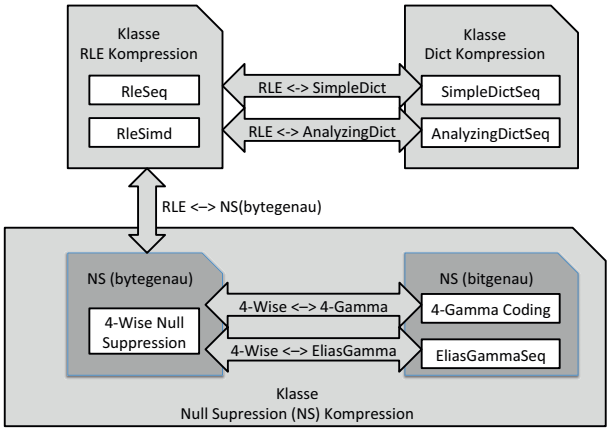


Abbildung 3: Überblick über den aktuellen Stand unserer Arbeit. Zu sehen sind Kompressionsverfahren (weiße Kästen) eingeordnet in Klassen (graue Kästen), sowie von uns entwickelte Transformationsverfahren (Pfeile). Die in diesem Artikel näher beschriebenen Verfahren sind *RleSimd* und die Transformation der Daten aus dem Kompressionsschema *AnalyzingDict* in das Schema *RLE*.

Bei den Verfahren, deren Name mit *„Seq“* endet, handelt es sich um sequenzielle Algorithmen. Analog dazu deutet das Suffix *„Simd“* auf einen mit Hilfe von SIMD-Instruktionen parallelisierten Algorithmus hin. Zwischen den betrachteten Kompressionsverfahren haben wir bereits mehrere Transformationsverfahren entwickelt, wie es in Abbildung 3 dargestellt ist. Dabei ist zwischen allen bisher betrachteten Paaren von Kompressionsverfahren eine Transformation in beide Richtungen möglich. Im Folgenden sollen exemplarisch unsere effiziente Implementierung von *RleSimd* sowie das Transformationsverfahren *Rle2AnalyzingDict* näher vorgestellt werden.

4.1 RLE-Kompression

Die grundlegende Idee von Run Length Encoding (RLE) besteht darin, die unkomprimierten Eingabedaten in *Läufe* einzuteilen. Ein Lauf ist eine ununterbrochene Aufeinanderfolge gleicher Werte, die durch einen Laufwert und eine Lauflänge charakterisiert ist. Jeder Lauf gibt somit an, welcher Wert wie oft wiederholt wird. Im Folgenden beschreiben wir das Format der komprimierten Daten sowie unseren vektorisierten Algorithmus *RleSimd*.

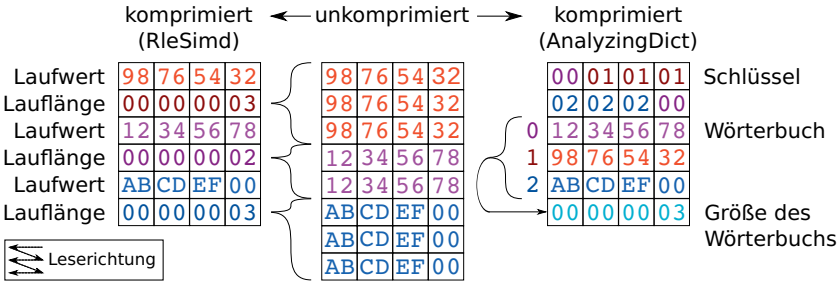


Abbildung 4: Eine beispielhafte Darstellung von unkomprimierten Daten und zugehörigen komprimierten Daten der Kompressionsverfahren RleSimd (links) und AnalyzingDict (rechts). Die Darstellung der Werte erfolgt in hexadezimaler Notation. Speicheradressen wachsen von rechts nach links und von oben nach unten, somit steht bei den unkomprimierten Daten in jeder Zeile ein 32-Bit-Integer im Little-Endian-Format

Komprimiertes Datenformat von RleSimd

Die mittels RleSimd komprimierten Daten sind eine Folge von 32-Bit-Integers, wie ebenfalls beim sequenziellen Verfahren. An allen geradzahligen Positionen (beginnend mit 0) befindet sich ein Laufwert, an allen ungeradzahligen Positionen eine Lauflänge. Die Reihenfolge der Läufe in den Originaldaten bleibt in den komprimierten Daten erhalten. Die linke Seite der Abbildung 4 zeigt ein Beispiel für mit RleSimd komprimierte Daten.

Algorithmus RleSimd

Kompression: Ein sequenzieller Algorithmus für RLE ist trivial implementierbar. Um jedoch die Performanz der Kompression zu optimieren, ist es notwendig, von den Möglichkeiten moderner Prozessoren bewusst Gebrauch zu machen. Dabei stellt insbesondere die Vektorisierung mittels SIMD-Instruktionen eine vielversprechende Möglichkeit dar, wobei wir intrinsische Funktionen in der Implementierung verwenden, um SIMD-Instruktionen zu nutzen. Unser vektorisierter Algorithmus übertrifft klar die Kompressionsgeschwindigkeit einer sequenziellen Implementierung, wie wir in Abschnitt 5 zeigen.

Am Anfang des vektorisierten Algorithmus' steht der Eingabezeiger auf dem erstem Wert der unkomprimierten Eingabedaten. Der Algorithmus geht in den folgenden Schritten vor (an Abbildung 5 können ausgewählte Schritte des Algorithmus nachvollzogen werden):

(a) Aktuellen Eingabewert (32-Bit-Integer) laden. Dieser ist der aktuelle Laufwert und wird mittels der Funktion `_mm_set1_epi32()` auf die vier 32-Bit-Elemente eines 128-Bit-Vektorregisters verteilt (Abbildung 5 a). Wir gehen im Moment von einer Größe des Vektorregisters von 128-Bit. Einer Vergrößerung ist jederzeit möglich.

(b) Die nächsten vier Werte der Eingabedaten mittels der Funktion `_mm_loadu_si128()` in ein zweites Vektorregister laden (Abbildung 5 b).

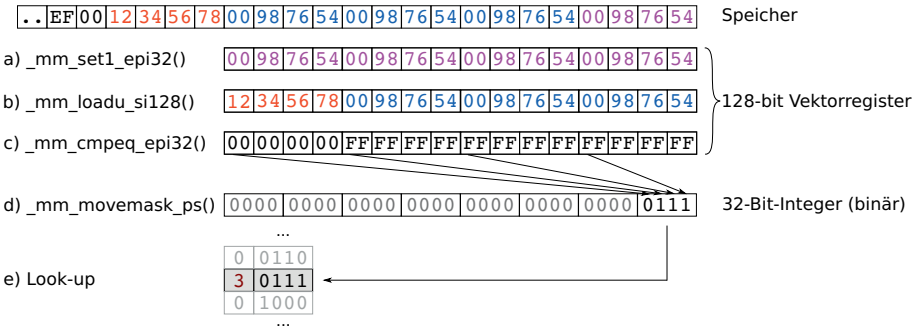


Abbildung 5: Darstellung ausgewählter Schritte des Kompressionsalgorithmus RleSimd an einem Beispiel. Alle Werte außer dem letzten sind in hexadezimaler Notation dargestellt. Adressen wachsen von rechts nach links. Oben: Ein Auszug des Speichers, der den Beginn der unkomprimierten Eingabedaten zeigt. Darunter: Registerinhalte nach Zwischenschritten des Algorithmus.

(c) Diese vier Werte mittels der Funktion `_mm_cmpeq_epi32()` parallel mit dem aktuellen Laufwert vergleichen (Abbildung 5 c). Das Ergebnis des parallelen Vergleichs liegt nun in einem Vektorregister vor. In jedem 32-Bit-Element dieses Vektorregisters sind entweder alle Bits gesetzt oder alle Bits nicht gesetzt, je nachdem, ob die korrespondierenden Elemente gleich waren oder nicht.

(d) Um dieses Ergebnis nutzen zu können, werden die höchstwertigen Bits jedes 32-Bit-Elements extrahiert und zu einer 4-Bit-Maske zusammengefügt (Abbildung 5 d). Dazu verwenden wir die Funktion `_mm_movemask_ps()`. Diese interpretiert die Elemente eines Vektorregisters jedoch als Single-Precision-Gleitkommazahlen, weshalb zunächst eine Typkonvertierung mittels `_mm_castsi128_ps()` erforderlich ist. Diese erzeugt allerdings keine Maschineninstruktionen und wirkt sich somit nicht auf die Performanz aus.

(e) Die entstandene Maske wird genutzt, um in einer zuvor erstellten Tabelle nachzuschlagen, um wie viele Elemente der aktuelle Lauf fortgesetzt wird.

(f) Falls der Lauf um weitere vier Elemente fortgesetzt wird, so ist das Ende des Laufs noch nicht bekannt. Der Eingabezeiger wird um vier Elemente weiter geschoben. Die Abarbeitung fährt mit Schritt (b) fort.
Falls der Lauf um weniger als vier Elemente fortgesetzt wird, so ist das Ende des Laufs bekannt. Der Laufwert und die Lauflänge werden an die Ausgabedaten angehängt. Der Eingabezeiger wird um so viele Elemente weiter geschoben, wie der aktuelle Lauf soeben fortgesetzt wurde. Damit steht er nun auf dem ersten Element des nächsten Laufs. Die Abarbeitung fährt mit Schritt (a) fort.

Kurz vor Erreichen des Endes der Eingabedaten sind unter Umständen nicht mehr ausreichend viele (d.h. weniger als vier) Eingabewerte für eine parallele Verarbeitung übrig. Diese restlichen Werte werden dann mit einem trivialen sequenziellen Algorithmus verarbeitet.

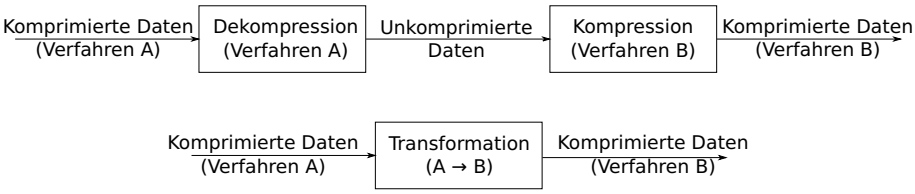


Abbildung 6: Oben: Klassischer Ansatz für einen Wechsel des Kompressionsverfahrens: Vollständige Dekompression gefolgt von vollständiger Rekompression. Unten: Unser Ansatz: Direkte Transformation ohne zwischenzeitliche vollständige Dekompression.

Initialisierung: In der Initialisierungsphase wird einmalig die Lookup-Tabelle für Schritt (e) des Kompressionsalgorithmus' aufgestellt. Diese Tabelle enthält 16 Einträge und ist mit den 4-Bit-Masken indiziert. Die Fortsetzungslänge zu einer Maske `m` ist die Anzahl der *„trailing ones"* in `m`. Diese kann berechnet werden als `table[m] = __builtin_ctz(~m)`, d.h. als die Anzahl der abschließenden Nullen der negierten Maske.

Dekompression: Für die Dekompression kommt ein einfacher, rein sequenzieller Algorithmus zum Einsatz: Bis das Ende der komprimierten Eingabedaten erreicht ist, werden jeweils Laufwert und Lauflänge des nächsten Laufs gelesen und der Laufwert entsprechend oft an das Ende der Ausgabedaten angehängt.

4.2 Transformation von RLE zu AnalyzingDict

Im Folgenden widmen wir uns der Transformation von komprimierten Daten eines Kompressionsverfahrens *A* in die eines anderen Verfahrens *B*. Im klassischen Fall ist dafür eine vollständige Dekompression gemäß Algorithmus *A* gefolgt von einer vollständigen Kompression gemäß Algorithmus *B* erforderlich. Unser Ziel besteht darin, die Transformation ohne zwischenzeitliche vollständige Dekompression der Daten zu ermöglichen. Stattdessen schlagen wir eine eng verzahnte De- und Rekompression vor, wie es beim *transient decompression*-Ansatz [CGK01] bereits für Operatoren gemacht wird. Idealerweise kann dabei ein Schreiben von unkomprimierten Daten in den Speicher vermieden werden. Abbildung 6 verdeutlicht den Unterschied zwischen beiden Alternativen.

Wir stellen *Rle2AnalyzingDict* als ein Beispiel, der von uns bereits konzipierten Transformationsverfahren (siehe Abbildung 3), vor. Dieses Verfahren erwartet Eingabedaten im komprimierten Format von RleSimd (siehe Abschnitt 4.1). Die Ausgabendaten haben das komprimierte Format des Kompressionsalgorithmus' *AnalyzingDictSeq*, welcher nun auch kurz vorgestellt wird.

Komprimiertes Datenformat von AnalyzingDict

Bei AnalyzingDict handelt es sich um ein wörterbuchbasiertes Verfahren. Das heißt, jedem unterschiedlichen Wert der unkomprimierten Daten wird ein Schlüssel zugeordnet, welcher der Position des zugehörigen Wertes im Wörterbuch entspricht. Die komprimierten Daten von AnalyzingDict lassen sich in zwei Abschnitte gliedern: (i) eine Folge von Wörterbuchschlüsseln (ein Schlüssel pro Originalwert) und (ii) das Wörterbuch selbst.

Schlüssel werden mit 0 beginnend fortlaufend vergeben, so dass sie bei d unterschiedlichen Werten im Intervall $[0, d-1]$ liegen. Beim Speichern der Schlüssel werden führende Null-Bytes weggelassen. Jedoch werden alle Schlüssel mit derselben Anzahl Bytes gespeichert. Bei d unterschiedlichen Werten ergeben sich $4 - (\_\_builtin\_clz((d-1)|1)/8)$ Bytes pro Schlüssel.

Das Wörterbuch enthält alle unterschiedlichen Werte der unkomprimierten Eingabedaten in aufsteigender Reihenfolge sowie die Anzahl der unterschiedlichen Werte als 32-Bit-Integers. Wir erwarten, dass die dadurch entstehende reihenfolgeerhaltende Kompression das Arbeiten auf den komprimierten Daten besonders begünstigt. Ein Beispiel für in diesem Format komprimierte Daten findet sich in der rechten Seite der Abbildung 4.

Kompressionsverfahren AnalyzingDictSeq

Der Kompressionsalgorithmus ist sequenziell und läuft in zwei Phasen ab. In der ersten Phase wird das Wörterbuch aufgestellt. Dazu wird einmal komplett über die Eingabedaten iteriert und jeder bis dahin nicht gesehene Wert dem Wörterbuch hinzugefügt. Anschließend wird das Wörterbuch sortiert und die Anzahl der notwendigen Bytes pro Schlüssel ermittelt. Während der Kompression dient das Wörterbuch dem Nachschlagen des Schlüssels zu einem Wert.

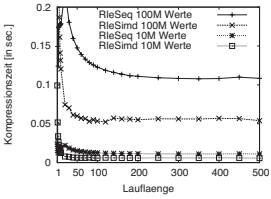
In der zweiten Phase findet die eigentliche Kompression der Eingabedaten statt. Nun wird erneut über die Eingabedaten iteriert. Dabei wird zu jedem Wert der zugehörige Schlüssel im Wörterbuch nachgeschlagen und mit der festgelegten Anzahl Bytes den Ausgabedaten angehängt. Zuletzt werden das Wörterbuch und dessen Größe an das Ende der Ausgabe geschrieben.

Im Zuge der Dekompression werden die Schlüssel wieder durch die zugehörigen Werte ersetzt. Wir verzichten an dieser Stelle auf eine ausführliche Beschreibung des Algorithmus', da dieser für die Betrachtung von Rle2AnalyzingDict nicht relevant ist.

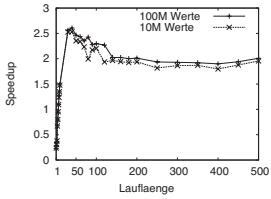
Transformationsverfahren Rle2AnalyzingDict

Der Transformationsalgorithmus *Rle2AnalyzingDict* ist weitgehend identisch mit dem Kompressionsalgorithmus von *AnalyzingDictSeq*. Der Ablauf gliedert sich in dieselben zwei Phasen, wobei hier auf eine vollständige Dekompression der Daten verzichtet werden kann.

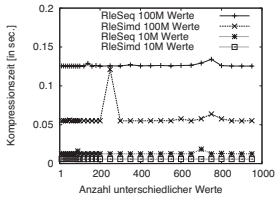
Bei der Erstellung des Wörterbuchs kann nun ein erheblicher Teil der Laufzeit eingespart werden. Während beim Arbeiten auf unkomprimierten Daten für *jeden Wert* geprüft wer-



(a) Lauflänge



(b) Speedup - Lauflänge



(c) Unterschiedliche Werte

Abbildung 7: Kompressionszeiten von *RleSeq* und *RleSimd* sowie Speedup von *RleSimd* gegenüber *RleSeq*. Die Ausgangsdaten in (a) und (b) setzen sich aus 10M bzw. 100M unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

den muss, ob dieser bereits im Wörterbuch enthalten ist, muss dies beim Arbeiten auf RLE-komprimierten Daten nur für *einen Wert pro Lauf* durchgeführt werden. Zudem müssen nur die Laufwerte betrachtet werden und die Lauflängen können übersprungen werden.

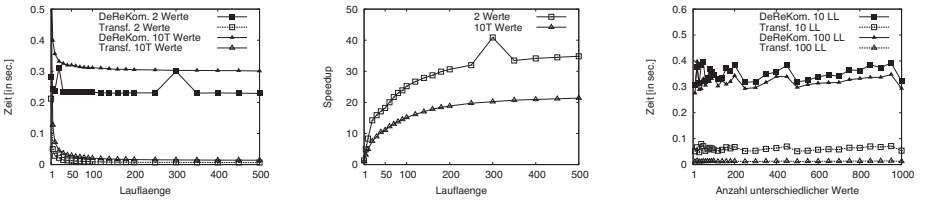
Auch die zweite Phase ist nun weniger aufwendig. Statt für *jeden Wert* der Eingabedaten den zugehörigen Schlüssel nachzuschlagen, muss dies nun wieder nur für *einen Wert pro Lauf* getan werden. Der erhaltene Schlüssel wird dann entsprechend der Lauflänge mehrmals an die Ausgabedaten angehängt.

5 Evaluierung

Unsere Algorithmen sind in C++ implementiert und benutzen SIMD-Instruktionen aus der Erweiterung SSE2 mittels intrinsischer Funktionen. Als Evaluierungssystem nutzen wir eine Intel Core i3-2350M CPU mit 4GB Hauptspeicher, wobei die Algorithmen mit gcc 4.7 und dem Optimierungsflag `-O3` kompiliert sind. Für die Experimente haben wir einen eigenen Datengenerator geschrieben, um 32-bit Integer-Werte mit vorgegebener Anzahl unterschiedlicher Werte und Lauflängen zu erzeugen. Die Verteilung der Daten erfolgt dabei gleichmäßig.

5.1 Sequenzielle vs. parallele RLE-Kompression

Als Erstes vergleichen wir die sequenzielle versus unsere SIMD-fähige RLE-Kompression. In Abbildung 7 ist dabei Folgendes zu erkennen: (1) dass sowohl für *RleSeq* als auch für *RleSimd* die benötigte Zeit mit steigender Lauflänge sinkt, (2) dass die Kompressionszeiten beider Algorithmen unabhängig von der Anzahl der unterschiedlichen Werte sind und (3) dass *RleSimd* unabhängig von der Größe der Ausgangsdaten ab einer Lauflänge von circa 8 schneller komprimiert als RleSeq. Die Speedups der Kompression mit *RleSimd* gegenüber der mit *RleSeq* sind in Abbildung 7 (b) dargestellt. Wie zu erwarten war, ist die parallele Variante effizienter als die sequenzielle Implementierung. Die Opti-



(a) Lauflänge (b) Speedup - Lauflänge (c) Unterschiedlicher Werte

Abbildung 8: Laufzeiten der Transformation (Transf.) und der vollständigen De- und Rekompression (DeReKom.) sowie Speedups der Transformation gegenüber der vollständigen De- und Rekompression. Die Ausgangsdaten in (a) und (b) setzen sich aus 2 bzw. 10.000 unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge (LL = Lauflänge) 10 bzw. 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

mierung der leichtgewichtigen Kompressionsverfahren durch Parallelisierung kann einen erheblichen Performanzgewinn bringen, der gerade mit dem Blick auf unsere anvisierte *ausgewogene Anfrageverarbeitung* in Datenbanksystemen besonders attraktiv ist.

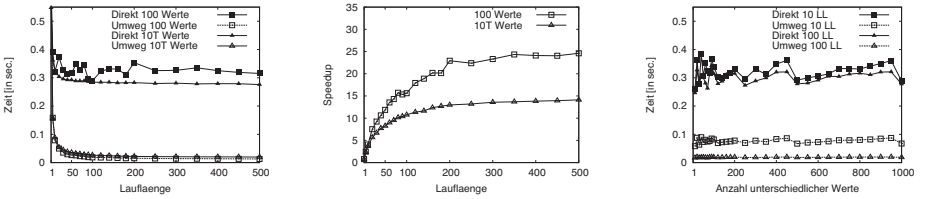
5.2 Transformation vs. vollständige De- und Rekompression

Als Nächstes widmen wir uns dem Wechsel des Kompressionsschemas, am Beispiel des Wechsels von RLE zu AnalyzingDict. Wir vergleichen die dafür benötigte Zeit bei Nutzung von (1) der direkten Transformation mit dem Algorithmus *Rle2AnalyzingDict* und (2) der vollständigen Dekompression mit *RleSimd* und Rekompression mit *AnalyzingDictSeq*. In Abbildung 8 ist klar zu erkennen, dass die direkte Transformation unabhängig von der Lauflänge und der Anzahl unterschiedlicher Werte deutlich weniger Zeit benötigt als die vollständige De- und Rekompression. Die erreichten Speedups der Transformation gegenüber der vollständigen De- und Rekompression sind in Abbildung 8 (b) dargestellt.

Wie aus diesem Experiment ersichtlich wird, ist der Transformationsansatz für die Änderung des Kompressionsschemas in diesem Szenario wesentlich effizienter als die vollständige Dekompression gefolgt von einer vollständigen Kompression. Für uns ist das ein Indikator dafür, dass derartige Transformationsverfahren sinnvoll sind und für die Anfrageverarbeitung betrachtet werden müssen. Dieser operationale Aspekt (siehe Abschnitt 3.2) wird als Nächstes von uns untersucht.

5.3 Transformations-beschleunigte Kompression

Eine weitere mögliche Anwendung der Transformationsalgorithmen sehen wir in der Beschleunigung der Kompression in ein Schema *B*, indem zunächst die Daten mittels eines Verfahrens *A* komprimiert und anschließend von dort nach *B* transformiert werden. Wir



(a) Lauflänge (b) Speedup - Lauflänge (c) Unterschiedlicher Werte

Abbildung 9: Laufzeiten der direkten Kompression (Direkt) nach *AnalyzingDict* und der Kompression über den Umweg *RleSimd* sowie Speedups der Kompression über den Umweg gegenüber der direkten Kompression. Die Ausgangsdaten in (a) und (b) setzen sich aus 100 bzw. 10.000 unterschiedlichen Integer-Werten zusammen und die Lauflänge wird variiert. Die Ausgangsdaten in (c) setzen sich aus Läufen der Länge (LL = Lauflänge) 10 bzw. 100 zusammen und die Anzahl der unterschiedlichen Integer-Werte wird variiert.

demonstrieren die prinzipielle Machbarkeit dieser Idee am Beispiel der Kompression in das Format *AnalyzingDictSeq*. Zum Einen wird *AnalyzingDictSeq* direkt ausgeführt und zum Anderen über den „Umweg“ *RleSimd* und *Rle2AnalyzingDict*.

Die Ergebnisse sind in der Abbildung 9 dargestellt. Wie zu erkennen ist, wird für alle Lauflängen außer 1 und beliebige Anzahlen unterschiedlicher Werte die Kompression über den Umweg erheblich schneller als auf dem direkten Weg. Die erreichten Speedups der Kompression über den Umweg gegenüber der direkten Kompression sind in der Abbildung 9(b) illustriert.

6 Zusammenfassung

In Hauptspeicher-zentrischen Architekturansätzen für Datenbanksysteme müssen für die Anfrageverarbeitung sowohl die Basisrelationen als auch die Zwischenergebnisse im Hauptspeicher gehalten werden. Zudem ist der Aufwand, Zwischenergebnisse zu generieren äquivalent zum Aufwand, Änderungen an den Basisrelationen durchzuführen. Daher sollten die Zwischenergebnisse gesondert betrachtet werden, wobei sich zwei orthogonale Optimierungsansätze anbieten. Auf der einen Seite sollten Zwischenergebnisse nicht mehr z.B. durch entsprechend angepasste Code-Generierung [Neu11] oder durch den Einsatz kooperativer Operatoren [KSHL13] produziert werden. Auf der anderen Seite sollten Zwischenergebnisse (wenn sie z.B. nicht vermeidbar sind) so organisiert werden, dass eine effiziente Weiterverarbeitung ermöglicht wird. Für den letzten Bereich schlagen wir den durchgängigen Einsatz leichtgewichtiger Kompressionsverfahren für Zwischenergebnisse vor und haben unsere Vision einer *ausgewogenen Anfrageverarbeitung auf Basis komprimierter Zwischenergebnisse* vorgestellt. Neben dem Überblick über unsere Vision haben wir ebenfalls die wissenschaftlichen Herausforderungen skizziert und den Stand der Technik umfassend aufgearbeitet. Darüber hinaus haben wir erste Ansätze für die Optimierung von leichtgewichtigen Kompressionsverfahren vorgestellt und erste Transformationsverfahren präsentiert, die aus Sicht der Anfrageverarbeitung eventuell sinnvoll sind, um einen

effizienten Wechsel des Kompressionsverfahrens durchzuführen. Des Weiteren hat unsere Evaluierung gezeigt, dass derartige Optimierungen einen erheblichen Speedup bringen.

Literatur

[ALM96] Gennady Antoshenkov, David B. Lomet und James Murray. Order Preserving Compression. In *ICDE Conference*, Seiten 655–663, 1996.

[AMF06] Daniel Abadi, Samuel Madden und Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, Seiten 671–682, 2006.

[AYJ00] Sihem Amer-Yahia und Theodore Johnson. Optimizing Queries on Compressed Bitmaps. In *VLDB Conference*, Seiten 329–338, 2000.

[Bas85] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. *IEEE Transactions on Software Engineering*, 11(10):1047–1058, 1985.

[BB07] Jeff Berger und Paolo Bruni. Index Compression with DB2 9 for z/OS. http://www.redbooks.ibm.com/redpapers/pdfs/redp4345.pdf, 2007.

[BCE77] Michael W. Blasgen, Richard G. Casey und Kapali P. Eswaran. An encoding method for multifield sorting and indexing. *Communications ACM*, 20(11):874–878, 1977.

[BHF09] Carsten Binnig, Stefan Hildenbrand und Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD Conference*, Seiten 283–296, 2009.

[BLM+09] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George A. Mihaila, Ken Ross, Sherman Lau, Cathy McCarthur, Zoltan Toth und Reza Sherkat. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2):1462–1473, 2009.

[BMK99] Peter A. Boncz, Stefan Manegold und Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB Conference*, Seiten 54–65, 1999.

[BU77] Rudolf Bayer und Karl Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, 1977.

[CGK00] Zhiyuan Chen, Johannes Gehrke und Flip Korn. Squeezing the Most Out of Relational Database Systems. In *ICDE Conference*, Seite 81, 2000.

[CGK01] Zhiyuan Chen, Johannes Gehrke und Flip Korn. Query optimization in compressed database systems. *SIGMOD Record*, 30(2):271–282, 2001.

[CIW84] John G. Cleary, Ian und Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Goy83] Pankaj Goyal. Coding methods for text string search on compressed databases. *Inf. Syst.*, 8(3):231–233, 1983.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan und Uri Shaft. Compressing Relations and Indexes. In *ICDE Conference*, Seiten 370–379, 1998.

[GS91] G. Graefe und L.D. Shapiro. Data compression and database performance. In *Applied Computing*, Seiten 22–27, Apr 1991.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey und E. Shekita. Starburst mid-flight: as the dust clears [database project]. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):143 –160, Marz 1990.

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart und David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD Conference*, Seiten 389–400, 2007.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[IW94] Balakrishna R. Iyer und David Wilhite. Data Compression Support in Databases. In *VLDB Conference*, Seiten 695–704, 1994.

[Joh99] Theodore Johnson. Performance Measurements of Compressed Bitmap Indices. In *VLDB Conference*, Seiten 278–289, 1999.

[KSHL13] Thomas Kissinger, Benjamin Schlegel, Dirk Habich und Wolfgang Lehner. QPPT: Query Processing on Prefix Trees. In *CIDR 2013*, 2013.

[LC86] Tobin J. Lehman und Michael J. Carey. Query processing in main memory database management systems. In *SIGMOD Conference*, Seiten 239–250, 1986.

[LLR06] Thomas Legler, Wolfgang Lehner und Andrew Ross. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB Conference*, Seiten 1059–1068, 2006.

[LZ76] A Lempel und J Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[MF04] Roger MacNicol und Blaine French. Sybase IQ multiplex - designed for analytics. In *VLDB Conference*, Seiten 1227–1230, 2004.

[MLL13] Christopher M. Mullins, Lipyeow Lim und Christian A. Lang. Query-aware Compression of Join Results. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT 2013)*, Seiten 29–40, 2013.

[MZ92] Alistair Moffat und Justin Zobel. Parameterised compression for sparse bitmaps. In *SIGIR Conference*, Seiten 274–285, 1992.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[PP03] Meikel Poess und Dmitry Potapov. Data compression in Oracle. In *VLDB Conference*, Seiten 937–947, 2003.

[QRR+08] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas und Guy M. Lohman. Main-memory scan sharing for multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[Reg81] Hassan K. Reghbati. An Overview of Data Compression Techniques. *IEEE Computer*, 14(4):71–75, 1981.

[RH93] Mark A. Roth und Scott J. Van Horn. Database Compression. *SIGMOD Record*, 22(3):31–39, 1993.

[RHS95] Gautam Ray, Jayant R. Haritsa und S. Seshadri. Database Compression: A Performance Enhancement Tool. In *COMAD Conference*, 1995.

[RS06] Vijayshankar Raman und Garret Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *VLDB Conference*, Seiten 858–869, 2006.

[RSQ+08] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang und Richard Sidle. Constant-Time Query Processing. In *ICDE Conference*, Seiten 60–69, 2008.

[SDF+00] Jagannathan Srinivasan, Souripriya Das, Chuck Freiwald, Eugene Inseok Chong, Mahesh Jagannath, Aravind Yalamanchi, Ramkumar Krishnan, Anh-Tuan Tran, Samuel DeFazio und Jayanta Banerjee. Oracle8i Index-Organized Table and Its Application to New Domains. In *VLDB Conference*, Seiten 285–296, 2000.

[SGL10] Benjamin Schlegel, Rainer Gemulla und Wolfgang Lehner. Fast Integer Compression using SIMD Instructions. In *DaMoN*, 2010.

[Sto88] James A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., New York, NY, USA, 1988.

[Tro03] Andrew Trotman. Compressing Inverted Files. *Inf. Retr.*, 6(1):5–19, 2003.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer und Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[WNC87] Ian H. Witten, Radford M. Neal und John G. Cleary. Arithmetic coding for data compression. *Communications ACM*, 30(6):520–540, 1987.

[WOS06] Kesheng Wu, Ekow J. Otoo und Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier und Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[YZ03] John Yiannis und Justin Zobel. External sorting with on-the-fly compression. In *BNCOD Conference*, Seiten 115–130, 2003.

[YZ07] John Yiannis und Justin Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, 2007.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes und Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE Conference*, Seite 59, 2006.

[ZIL93] A. Zandi, B. Iyer und G. Langdon. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, Seiten 330 –339, 1993.

[ZL77] Jacob Ziv und Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

SPARQling Pig – Processing Linked Data with Pig Latin

Stefan Hagedorn[1], Katja Hose[2], Kai-Uwe Sattler[1]

[1] Technische Universität Ilmenau, Ilmenau, Germany
{first.last}@tu-ilmenau.de
[2] Aalborg University, Aalborg, Denmark
khose@cs.aau.dk

Abstract: In recent years, dataflow languages such as Pig Latin have emerged as flexible and powerful tools for handling complex analysis tasks on big data. These languages support schema flexibility as well as common programming patterns such as iteration. They offer extensibility through user-defined functions while running on top of scalable distributed platforms. In doing so, these languages enable analytical tasks while avoiding the limitations of classical query languages such as SQL and SPARQL. However, the tuple-oriented view of general-purpose languages like Pig does not match very well the specifics of modern datasets available on the Web, which often use the RDF data model. Graph patterns, for instance, are one of the core concepts of SPARQL but have to be formulated as explicit joins, which burdens the user with the details of efficient query processing strategies. In this paper, we address this problem by proposing extensions to Pig that deal with linked data in RDF to bridge the gap between Pig and SPARQL for analytics. These extensions are realized by a set of user-defined functions and rewriting rules, still allowing to compile the enhanced Pig scripts to plain MapReduce programs. For all proposed extensions, we discuss possible rewriting strategies and present results from an experimental evaluation.

1 Introduction

Processing and analyzing RDF [W3C04] data and particularly Linked (Open) Data [BL06] often requires operations going beyond the capabilities of classic query languages such as SPARQL [W3C08]. Although the most recent W3C recommendations for SPARQL support federated queries and aggregate functions, typical data preparation and cleaning steps as well as more advanced analytical tasks cannot easily be implemented directly in SPARQL or any similar query language.

Consider for example a user who wants to obtain information about events in all available categories (festival, jazz, orchestra, rock, etc.) that take place in the five biggest cities in the world. Even if the user has full access to the event dataset, determining which 5 cities are the biggest in the world requires access to another remote dataset. Answering such a query further requires expensive grouping and can therefore easily become expensive to evaluate in a triple store. Furthermore, integrating non-RDF sources, e.g., CSV files or complex tasks such as clustering the events on their geographic location or data cleaning tasks are not possible in SPARQL engines.

On the other hand, with the wide acceptance of the MapReduce platform [DG08] declarative dataflow languages such as Pig [ORS+08] or Jaql [BEG+11] have gained much atten-

tion. They provide a rich set of data processing operations, transparent parallelization to exploit data parallelism, can deal with flexible schemas, and are easy to learn even for non-specialists. Furthermore, these languages can easily be extended by user-defined functions enabling custom processing. Finally, by targeting MapReduce as execution environment, dataflow scripts are deployable to large-scale clusters for analyzing big datasets.

Although Pig supports a nested data model which is based on bags of tuples that in turn can contain bags, a tuple-oriented view results in several limitations and shortcomings:

- While SPARQL provides basic graph patterns (BGP) as a core concept of query formulation, such query patterns have to be expressed as self joins in a tuple-oriented language. Unfortunately, this is a quite expensive operation; first, because it is implemented in Pig by a so-called `COGROUP` operator representing a generalized form of join and grouping that is mapped to a combination of map and reduce tasks. The second reason is that self joins in Pig require to load the data twice, i.e., once for each branch of a join. Furthermore, the user is responsible for formulating efficient scripts, e.g., in terms of join order or join implementation.
- Pig is targeting the MapReduce platform and therefore works best if the data can be loaded from HDFS [Fou09]. However, processing Linked Data often requires to access and retrieve data from remote (SPARQL) endpoints – which is usually addressed by federated SPARQL engines. Depending on the size of the remote dataset, the capabilities of the endpoint, and the frequency of script execution, it might be useful to download and materialize the dataset before running the Pig script or to fetch (and cache) the remote data on demand. In both cases, this has to be explicitly implemented in the scripts.
- Pig provides a flexible nested data model that offers several alternatives for representing groups of triples or RDF statements. This includes not only the representation while processing the data but also the storage format in HDFS files. However, the choice of the most efficient format also depends on the workload and is hence again the responsibility of the user.

To overcome these limitations, this paper presents Pig language extensions adding SPARQL-like features that are particularly useful for implementing analytical processing tasks over Linked Data. The main goal of our work is to provide operators on a level of abstraction that lifts the burden of choosing the most efficient strategy and operator implementations from the user. These extensions include efficient support of BGPs and `FILTER`s as well as transparent access to remote Linked Open Data. Our contribution is twofold:

- We present a set of language extensions for making Pig a language suitable for analytical Linked Data processing. We discuss possible strategies for implementing these extensions in Pig.
- We report results from an experimental evaluation of the different strategies. In lack of a cost model and cost-based Pig compilers, these results can be used as a foundation to implement rewriting heuristics in Pig.

The work described in this paper is embedded in a larger vision of a platform for (Linked) Open Data analytics (LODHub [HS14]) that allows to publish, share, and analyze Linked Datasets. In this context, Pig is used – under the hood of a visual dataflow designer – as the data processing language.

The remainder of the paper is structured as follows. After a discussion of related work in Sect. 2, we briefly describe the context of our work in Sect. 3 and derive requirements on Pig suitable extensions for Linked Data processing. The proposed extensions are then presented in Sect. 4, followed by a discussion of the necessary dataflow planning and compiling steps in Sect. 5. Results of the experimental evaluation of the implementation and rewriting strategies are reported in Sect. 6. Finally, Sect. 7 concludes this paper and outlines future work.

2 Related Work

With the continuous growth of RDF datasets, we experience scalability issues when using a single server to process SPARQL queries. Despite efficient centralized SPARQL engines for a single server installation, such as RDF-3X [NW08], literature has proposed several systems [KM14] that use parallel processing in a cluster of machines. The architectures used for this purpose differ between orchestrating multiple centralized RDF stores [GHS14], using key-value stores [ZYW+13], or building upon distributed file systems [SPZL11, ZCTW13].

Especially systems using a distributed file system make use of MapReduce [DG08] to process joins [HMM+11, PKT+13]. One of the key considerations is to minimize the number of MapReduce cycles while still computing the complete answer to the query. Instead of directly mapping SPARQL queries to MapReduce programs, Pig Latin [ORS+08] can be used as an intermediary step, i.e., SPARQL queries are mapped to Pig Latin, which in turn is compiled into MapReduce. One of the first approaches of this class was presented in [SPZL11] where the translation of SPARQL queries into Pig Latin scripts is discussed. In this work, RDF data is represented as plain tuples of three fields and the operators of the SPARQL algebra are mapped to sequences of Pig Latin commands. In contrast to this approach, our work aims at integrating SPARQL elements such as BGPs into Pig, keeping the dataflow language as the primary language for specifying data analytics pipelines. Furthermore, we support multiple RDF datasets and SPARQL endpoints within a single script. In addition, we discuss and exploit different (intermediate) data representation formats for RDF data together with possible optimization strategies depending on these data representations.

An alternative line of optimization techniques make use of algebraic optimization [RKA11] to minimize the number of MapReduce cycles on a system based on Pig in combination with an algebra [KRA11] that considers triple groups, e.g., all triples with the same subject are considered a group. In our work we follow a similar idea of representing RDF triples but generalize the approach and consider it as only one alternative strategy for deriving efficient MapReduce programs.

In comparison to processing of standard SPARQL queries, analytical queries pose additional challenges due to their special characteristics such as complexity, evaluation on

typically very large datasets, and long runtime [ADE+13, CGMR14]. Data cubes with dimensions and measures over RDF data can be defined using RDF compliant standards [EV12, W3C13]. On this basis, analytical queries can be formulated, evaluated, and optimized. Kotoulas et al. [KUBM12] propose an approach for query optimization that interleaves query execution and optimization and SPARQL queries are translated to Pig Latin.

In summary, existing systems have focused on RDF data organized in a single dataset that the user has complete control over. In contrast, this paper proposes a system that targets a broader perspective by considering analytical queries and Linked Data organized in multiple datasets and even remote sources that the user who issues the query does not have any control over.

3 Use Case: The LODHub Platform

The goal of LODHub [HS14] is to combine the functionalities of frameworks for Open Data management with infrastructure for storing and processing large sets of Linked (Open) Data. An elastic server infrastructure, e.g., hosted by IaaS (Infrastructure as a Service) providers such as Amazon EC2, will allow to start machines automatically as workload peaks occur. Queries and datasets can then be distributed among these machines transparently to the user.

LODHub is designed to encourage users to share their data with other users and thus contribute to the idea of Open Data. Still, users can also choose to keep their datasets private. Additionally, as users often tend to work in teams, the platform will allow to associate user accounts with organizations and to organize users in teams. When sharing datasets with other users or creating a new version of an existing dataset, provenance information will be kept to track the original author.

Data exploration features will help users to discover new knowledge by automatically extending their query results with information found not only in other available datasets within LODHub, but also from remote sources on the Web.

The integration of Pig [ORS+08] enables users to perform analytical tasks that are not possible with a pure SPARQL approach. Such tasks include data transformation, combining different sources with different formats, and other complex tasks for which no equivalent SPARQL operations exist.

To let users work with their data, LODHub provides an easy-to-use and intuitive graphical editor to create Pig scripts. With this editor, a user can can drag and drop operators and edit their parameters. Operators have ports for input and output, and by connecting the output port of one operator to the input port of another operator, a dataflow is created. Connecting all needed operators results in a dataflow graph, which must contain one or more `LOAD` operators and at least one `DUMP` or `STORE` operator to show the results on screen or to save them as a new dataset, respectively. Between those two types of operators, users can add as many operators as needed. The screenshot in Fig. 1 shows the graphical Pig editor as well as the Pig script compiled from the operator graph.

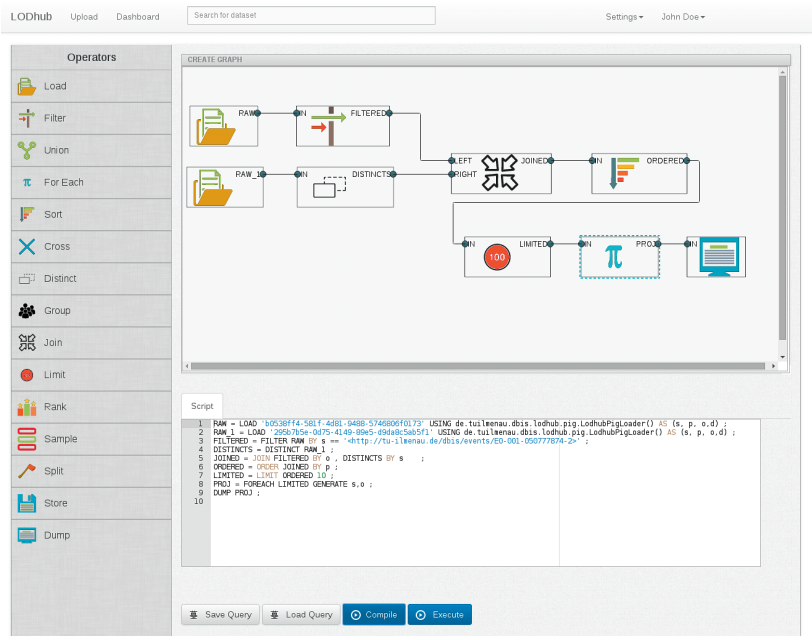


Figure 1: LODHub's visual dataflow designer for Pig scripts

In its current implementation, LODHub features almost all operators that Pig provides. Some of these operators, however, were modified to meet the challenges that arise when working with RDF data. Note, that LODHub is able to work not only with RDF data, but with any format for which a load function, e.g. as a UDF, exists.

The following scenario illustrates an example scenario and an analytical task that is supported by LODHub. Assume there exists an RDF dataset with information about events. These events may be concerts, sport events, etc., or even some sensor readings that have been logged into a file. Among other information each event is described by a title, a description of the event or sensor reading, a category (i.e., "sport", "music", or "temperature"), and a position of the event in the form of lat/long coordinates.

Consider the following excerpt from a dataset (in total ca. 6.5 million events and more than 53 million statements) that were extracted from the website `http://eventful.com` and transformed into RDF (for simplicity, we abbreviate namespaces and event IDs):

```
<ev:event1> <ev:#title> "Metallica" .
<ev:event1> <ev:#start_time> "2012-08-17T18:00:00+02" .
<ev:event1> <ev:#venue_name> "Rexall Place" .
<ev:event1> <geo:#long> "-1.135E2" .
<ev:event1> <geo:#lat> "5.353E1" .
<ev:event1> <ev:#category> "music" .
```

Assume a user wants to find the number of events in each category for each of the five biggest cities in the world. To answer this question, we first need to find five biggest cities.

This information could for instance be extracted from the *remote* DBpedia [ABK+07] dataset, which contains information about cities including name, location, and population.

The next step is to find out, which event took place in which city. As the event dataset contains the coordinates in two separate RDF statements (longitude and latitude), a *self-join* is required to obtain the full coordinate. Now, using these coordinates, we can use an *external webservice* such as GeoNames [Wic] to obtain the name of the city that an event is located in. Once we have identified the corresponding city for each event, we can join the results obtained from the DBpedia dataset with our events dataset on the cities. This produces the events located in the 5 biggest cities.

This intermediate result now has to be joined again with our original event dataset to obtain the event category. We can group this join result on city and event category and then count the frequencies to produce the answer to the user's query.

Based on this example scenario, we identify several challenges to Pig operators:

Loading datasets. First, the dataset loader must be able to efficiently load datasets from the file system (HDFS [Fou09] in our framework). It should hide URIs and file names from the user as users should not be bothered with details such as the internal file locations. Furthermore, the dataset loader must handle the special data format that is optimized for Pig on RDF data (see Sect. 4 for details).

Basic Graph Pattern. SPARQL uses BGPs that are matched against the underlying dataset. Pig, however, does not support these BGPs natively. To allow users to easily work with RDF data, the Pig operators must be able to efficiently interpret BGPs, which may range from single statements to complex graph patterns involving multiple joins.

Self joins. RDF data is based on the concept of triples defining a relationship (property) between a subject and an object. Therefore, processing SPARQL queries requires to evaluate many self joins to process the BGPs defined in SPARQL queries. In contrast to triples stores, Pig (and Hadoop) was not designed for RDF data and assumes only very few joins within a query. This makes plain Pig and Hadoop very inefficient on RDF data, because performing self joins requires to load the same dataset twice. As an example, consider the previous scenario: to construct the coordinate pair for the events, the dataset has to be loaded twice in order to perform the self join. And later to add the category information, we again have to perform a join with the original event dataset.

Remote joins. One of the benefits of RDF is its distributed nature. Information can be distributed over many servers and through the links within the datasets, these datasets are inter-connected. This means that processing a query might include performing a join with a dataset on a remote server. As datasets can be large, a lot of information might have to be downloaded. To assure reasonable response times, query strategies for such remote joins have to implemented. In our scenario, we need to compute a join with the DBpedia dataset. Such a join can be performed in different ways depending on the remote dataset's size and the data needed to answer the query: the datasets can either be downloaded completely (if possible) or if a SPARQL endpoint is available, a SPARQL query can be sent to retrieve only relevant parts of the dataset, which probably results in only a small amount of data to download.

4 Pig Extensions for Linked Data Processing

In order to address the requirements discussed in Sect. 3, we have developed several extensions to the Pig Latin language which we introduce in the following. These extensions particularly deal with features for processing RDF data as well as defining analytical pipelines and can be classified into the following groups:

- conversion between RDF triples and the Pig data model for efficient processing,
- accessing linked (RDF) datasets,
- supporting basic graph patterns for users more familiar with the SPARQL style of query formulation.

`TUPLIFY` – Tuple construction. Although the basic triple structure of RDF data in the form of subject s, predicate p, and object o is very flexible and allows to represent arbitrary structured data including graphs, it is not the optimal choice for efficient analytics. The main reason is that reconstructing complex objects from multiple triples requires (self) joins, which are very expensive particularly in MapReduce-based execution models such as the model underlying Pig. On the other hand, a rigid relational structure with a predefined schema, where a set of triples with the same subject is represented by $(s, o_1, o_2, \ldots, o_n)$ for the schema $(s, p_1, p_2, \ldots p_n)$, is often not flexible enough for RDF data. Fortunately, Pig provides a flexible nested data model with tuples, bags, and maps. Thus, tuple construction is an important operation necessary to process RDF data in Pig efficiently.

A first possible representation is to use maps for the predicate-object pairs, i.e., $(s, \{p_1 \rightarrow o_1, p2 \rightarrow o_2, \ldots\})$. However, this prohibits the use of the same predicate multiple times for a given tuple. Therefore, an alternative approach uses bags instead of maps for the predicate-object pairs, i.e., a schema {`subject:bytearray, stmts:{(predicate:bytearray, object:bytearray)}`}. This can further be generalized by allowing to group triples on any of the three components, e.g., on the predicate so that we obtain {`predicate:bytearray, stmts:{(subject:bytearray, object:bytearray)}`}. Which of these representations is the best choice depends on the operations that shall be applied on the data.

The details of tuple reconstruction are hidden by a `TUPLIFY` operator which is mapped to different implementations. If the input is a plain bag of triples, it simply groups them as follows:

```
triple_groups = FOREACH (GROUP triples BY subject)
   GENERATE group as subject,
            triples.(predicate, object) AS stmts;
```

`RDFLoad` – Accessing datasets. For accessing linked datasets we have to distinguish between local and remote access. The most straightforward way of local access is to load textual files in the Notation 3 (N3) format from HDFS, which is natively supported by Pig's `LOAD` operator. A more efficient storage scheme both in terms of space consumption and read costs is to leverage triple groups by using adjacency lists, i.e., a set of triples

$\{(s_1, p_1, o_1), (s_1, p_2, o_2), \ldots, (s_1, p_n, o_n)\}$ is represented by $(s_1, \{(p_1, o_1), \ldots, (p_n, o_n)\}$. Because the Pig data model supports bags, maps, and nested tuples, textual files representing this structure can be loaded directly into Pig scripts. In addition, by using a user-defined function (UDF) for `LOAD` this principle can also be extended to block-oriented binary files using dictionary and/or run-length encoding of the s, p, and o values.

Currently, we support the following strategies for accessing RDF data from HDFS:

- using Pig's native `LOAD` operator for N3 files. Because RDF literals can contain whitespaces, a UDF (`RDFize`) for tokenizing text lines containing triples into RDF components is used as part of a macro:

```
REGISTER sparklingpig.jar;
DEFINE RDFFileLoad(file) RETURNS T {
    lines = LOAD '$file' AS (txt: chararray);
    $T = FOREACH lines
         GENERATE FLATTEN(pig.RDFize(txt))
         AS (subject, predicate, object);
}
triples = RDFFileLoad('rdf-data.nt'):
```

- using Pig's `LOAD` operator with the `BinStorage` function for triple groups:

```
rdf_tuples = LOAD 'rdf-data.dat' USING BinStorage() AS
   (subject: bytearray,
    stmts: (predicate: bytearray, object:bytearray));
```

In addition, RDF datasets may also be retrieved from remote SPARQL endpoints. For this purpose, we provide a UDF, called `SPARQLLoader`, that sends a given SPARQL query to the endpoint and transforms the result data (usually encoded in XML or JSON) into Pig tuples:

```
REGISTER sparklingpig.jar;
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader('SELECT * WHERE { ?s ?p ?o }')
      AS (subject, predicate, object);
```

Note that the latter opens up new options to deal with the retrieved data: depending on the size of the dataset, it could be useful to download and materialize the data in HDFS for reuse before starting the actual processing. Te keep the load at public SPARQL endpoints low, it is of course advisable to download database dumps when available and manually store them in HDFS.

Furthermore, depending on the user query, we can choose not to download all triples of the remote endpoint but only those triples that are necessary to answer the user's query. For instance, given a FILTER expression on the population in a user query, then the following script retrieves only triples encoding the population of cities:

```
REGISTER sparklingpig.jar;
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader(
           'CONSTRUCT {?s dbpedia:populationTotal ?pop }
           WHERE {?s rdf:type dbpedia:City .
                  ?s dbpedia:populationTotal ?pop}')
      AS (subject, predicate, object);
```

If metadata is available that describes the storage format of the data and indicates whether data from remote endpoints have already been downloaded, then the details of these different strategies can be encapsulated by a generic `RDFLoad` operator.

`FILTER` – Basic Graph Pattern. BGPs are sets of triple patterns that are used not only for simple filtering but also for joins. Translating BGPs in the style of relational query processing results in a number of expensive joins. Furthermore, the exact formulation depends on the triple/tuple layout of the data; for example, if triples are used then we need a `JOIN`, whereas for bag-based tuple structures, nested `FOREACH`/`FILTER` operations can be used. Consider the following simple BGP as an example:

```
{ ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?o1 .
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o2 }
```

Based on a triple schema in Pig, i.e., {`subject:bytearray, predicate:bytearray, object:bytearray` }, the BGP could be implemented by:

```
triples1 = RDFLoader('rdf-data.nt');
triples2 = RDFLoader('rdf-data.nt');
result = JOIN triples1 BY (subject), triples2 BY (subject);
```

Note that for self joins Pig requires to load the data twice as in the example above. Of course, this would not be necessary if the second pattern stemmed from another dataset.

In contrast, if we have constructed RDF tuples using the schema {`subject:bytearray, stmts:{(predicate: bytearray,object: bytearray)}`} before, i.e., creating a bag of pairs (predicate, object) belonging to a subject, then we can implement the BGP by:

```
tmp = FOREACH rdf_tuples {
   r1 = FILTER stmts
        BY (predicate == '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>');
   r2 = FILTER stmts
        BY (predicate == '<http://www.w3.org/2003/01/geo/wgs84_pos#long>');
   GENERATE *, COUNT(r1) AS cnt1, COUNT(r2) AS cnt2;
};
result = FILTER tmp BY cnt1 > 0 AND cnt2 > 0;
```

For convenience, we hide the details of implementing BGP processing by an extended `FILTER` operator which accepts a BGP as part of the `BY` clause, but have to translate this expression accordingly:

```
result = FILTER triples BY
         { ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?o1 .
           ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?o2 };
```

5 Pig Planning and Compiling

In the previous section we described several extensions to Pig Latin to process Linked Data. To execute scripts involving these extensions, we need to transform them into plain Pig scripts. This transformation is implemented as a three aspects: planning, rewriting, and UDFs. The remainder of this section discusses these aspects in more detail.

Planning. Given a user query, we can often use metadata to rewrite the query into a more efficient version. Since our extended Pig, for instance, allows access to local as well as remote datasets, a query might contain `LOAD` operations on a remote SPARQL endpoint. Given the information that the dataset was already downloaded and materialized while executing another query, we can rewrite the original query into one that uses a local dataset instead.

To identify these cases, we need to collect metadata and statistics, e.g., a mapping `(URL, query)` $\rightarrow$ `HDFS path` that we can use to decide whether a remote dataset is already available locally. As an example, consider the `LOAD` operation from Sect. 4:

```
raw = LOAD 'http://endpoint.org:8080/sparql'
      USING SPARQLLoader('SELECT * WHERE { ?s ?p ?o }')
      AS (subject, predicate, object);
```

If a mapping `(http://endpoint.org:8080/sparql, SELECT * WHERE {?s ?p ?o})` $\rightarrow$ `hdfs:///rdf-data.nt` exists, then we can rewrite the expression as

```
raw = RDFFileLoad('hdfs:///rdf-data.nt');
```

which can obviously be executed more efficiently by avoiding expensive data exchange over the Web.

Note that this cannot be achieved in a pure Pig environment because of the missing option to generate and store meta information. A framework like LODHub, however, could easily store such information and reuse data across user queries.

In addition to the above mentioned mappings, LODHub could also store information about the file format, i.e., if the content of a file is structured as plain triples or as triple groups, as well as additional statistics (e.g., VoID [W3C10] statistics) about the downloaded datasets, which will allow for further optimization.

Rewriting. Though UDFs provide a flexible mechanism to extend Pig with additional functionality, we decided not to implement all extensions as UDFs because transforming the extended Pig expressions to plain Pig allows us to use all optimizations that Pig already comes with. Instead, we decided to implement BGP support via rewriting rules, i.e., before execution the Pig script is passed to a rewriter that analyzes the code and uses rewriting rules to transform extended Pig statements into plain Pig statements. These rules cover various input formats, BGPs, and operations. An example for such a rewriting was already given in Sect. 4 for the extended filter operation. The input format defines the structure of the data stream, i.e., if they are plain tuples from an N3 file or an external SPARQL endpoint, or if the stream is already in a triple group representation as introduced in Sect. 4. In the following, we present and discuss these rewriting rules in detail.

We use the notation $alias_{format}^{(schema)}$ to describe the characteristics of the input and output data of an operator. The *alias* is the Pig alias of the input or output data, *format* describes the format of the data, i.e., if it is plain triples or triple groups. In the former case we write *plain* and in the latter case we write *group*. The *schema* denotes the schema of the data, e.g., `(subject, predicate, object)` for plain triples or `(subject, {(predicate, object)})` for triple groups. For ease of presentation, the following rules use the subject (s) as a default to denote grouping – nevertheless, the rules hold for other groupings as well.

(T1) If the data is available as plain triples, it can be tuplified using the TUPLIFY operator. It is the task of the optimizer to decide whether or not to apply this rule in order to create triple groups for a given input.

$alias^{(s,p,o)}_{plain}$
$\Rightarrow$
$out^{(s,\{(p,o)\})}_{group}$ = TUPLIFY(alias) = FOREACH (GROUP *alias* BY s)
GENERATE group as s, *alias*.(p, o) AS stmts;

(R1) If the RDFLoad operator is used with a reference to a remote endpoint ($uri.protocol = 'http'$), replace the operator with the UDF SPARQLLoader as introduced in Sect. 4.

out = RDFLoad('*uri*') AS (*schema*);
$\Rightarrow$
$out^{(schema)}_{plain}$ = LOAD '*uri*'
USING pig.SPARQLLoader(
'SELECT * WHERE { ?s ?p ?o }')
AS (*schema*);

(R2) As already outlined in Sect. 4, we can use a restrictive query to reduce the amount of triples fetched from the SPARQL endpoints. The necessary knowledge can be drawn from FILTER operations that use BGPs.

out = RDFLoad('*uri*') AS (*schema*);
out2 = FILTER *out* BY {BGP_1 . BGP_2};
$\Rightarrow$
$out^{(schema)}_{plain}$ = LOAD '*uri*'
USING pig.SPARQLLoader(
'CONSTRUCT * WHERE { BGP_1 }')
AS (*schema*);
$out2^{(schema)}_{plain}$ = FILTER $out^{(schema)}_{plain}$ BY { BGP_2 };

BGP_1 and BGP_2 represent basic graph patterns, e.g., star joins. The (sub)set of the BGP that restricts the remote data can be used to define the query that is sent to the SPARQL endpoint. Often, the FILTER's BGP cannot completely be used in the query. In this case, the remaining part of the BGP still has to be applied in a FILTER. Note that the LOAD does not have to be directly followed by the FILTER to match this rule. Rather, the output of the LOAD has to be passed as input to the FILTER transitively.

(L1) To load a local dataset, the URI uses the *hdfs* protocol (*uri.protocol* = *'hdfs'*). In this case, the RDFLoad operation can be replaced by the RDFFileLoad macro. If the dataset has been saved as plain triples, the output of the loader are plain triples, too.

out = RDFLoad('*uri*');
$\Rightarrow$
$out^{(s,p,o)}_{plain}$ = RDFFileLoad('*uri*');

(L2) Loading local datasets that have been saved in the triple group format produces an output with the triple group schema. Consider the output was grouped by subject (s), the RDFLoad operation will be rewritten as:

```
out = RDFLoad('uri');
⇒
out_group^(s,{(p,o)}) = LOAD 'uri' USING BinStorage()
                        AS (s:chararray, stmts:bag{
                          t:(p:chararray,o:chararray)
                        });
```

The resulting schema depends on which component (subject, predicate, or object) was used for grouping. This information can be stored in the meta data for each dataset.

(F1) A singe triple pattern may use only unbound variables. If such a triple pattern is used in a FILTER, it can be omitted since it has no effect:

```
out^(schema_1) = FILTER in^(schema_1) BY { ?s ?p ?o };
out2^(schema_2) = any_operator out^(schema_1);
⇒
out2^(schema_2) = any_operator in^(schema_1);
```

Note, that in this case, the format of the aliases does not matter.

(F2) Filter operators that use a single triple pattern with only one bound variable can be rewritten as a simple FILTER with one predicate.

```
out_plain^(s,p,o) = FILTER in_plain^(s,p,o) BY { ?s ?p 'value' };
⇒
out_plain^(s,p,o) = FILTER in_plain^(s,p,o) BY o == 'value';
```

Rewriting rules for one bound variable in the other two components (*subject* or *predicate*) are obviously analogously defined. Hence, we omit them here.

(F3) If the triple pattern of a filter contains two (ore more) bound variables, it is rewritten as a filter operation that connects both predicates by AND.

```
out_plain^(s,p,o) = FILTER in_plain^(s,p,o) BY { ?s 'value1' 'value2' };
⇒
out_plain^(s,p,o) = FILTER in_plain^(s,p,o)
                    BY p == 'value1' AND o == 'value2';
```

As in rule (F2) above, there are analogous rules for the bound variables in other (or all three) components.

(F4) If the input alias of the FILTER operator is in the triple group format and if the one bound variable is the grouping column, e.g., s, the following rule applies:

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { 'value' ?p ?o };
⇒
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY s == 'value';
```

(F5) If the bound variable of the filter triple pattern is not the grouping column, e.g., o, then the filter operation becomes more complex:

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { ?s ?p 'value' };
⇒
tmp_group^(s,{(p,o)},cnt) = FOREACH in_group^(s,{(p,o)}) {
  t = FILTER stms BY o == 'value';
  GENERATE *, COUNT(t) AS cnt;
};
out_group^(s,{(p,o)},cnt) = FILTER tmp_group^(s,{(p,o)},cnt) BY cnt > 0;
```

(F6) If the triple pattern contains two bound variables and neither of them is the grouping column, then both predicates can be connected by AND.

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { ?s 'v1' 'v2' };
⇒
tmp_group^(s,{(p,o)},cnt) = FOREACH in_group^(s,{(p,o)}) {
  t = FILTER stms BY p == 'v1' AND o == 'v2';
  GENERATE *, COUNT(t) AS cnt;
};
out_group^(s,{(p,o)},cnt) = FILTER tmp_group^(s,{(p,o)},cnt) BY cnt > 0;
```

(F7) If the triple pattern contains two bound variables and one is the grouping column, it will be split into two filter operations. These can then further be processed by rules (F4) and (F5).

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { 'v1' ?p 'v2' };
⇒
tmp_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { 'v1' ?p ?o };
out_group^(s,{(p,o)},cnt) = FILTER tmp_group^(s,{(p,o)}) BY { ?s ?p 'v2' };
```

(F8) Analogously to rule (F7), if the input is in the triple group format and a filter triple pattern contains only bound variables, it will be replaced by two filter operations, which in turn can further be processed by rules (F4) and (F6).

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { 'v1' 'v2' 'v3' };
⇒
tmp_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)}) BY { 'v1' ?p ?o };
out_group^(s,{(p,o)},cnt) = FILTER tmp_group^(s,{(p,o)}) BY { ?s 'v2' 'v3' };
```

(F9) If a filter contains more than one triple pattern ($TP_1, ..., TP_N$), i.e. a BGP, and they do not form a join, they can be evaluated one after the other as an implicit AND. We leave it to the Pig engine to optimize the sequence of filters. In this case the format of the respective aliases again do not matter.

```
out^(schema) = FILTER in^(schema) BY { TP1 ... TPN };
⇒
f1^(schema) = FILTER in^(schema) BY { TP1 };
...
fN^(schema) = FILTER f(N-1)^(schema) BY { TPN };
out^(schema) = fN;
```

(J1) If the data is available in plain triple format and if the BGP of a filter expression forms a star join (i.e., the triple pattern $TP_1, ..., TP_N$ share the same variable in the same position, e.g., on subject), the dataset has to be loaded for each BGP participating in the join. Such joins may contain bound variables. Hence, we first filter the input, for these bound variables.

```
in_plain^(s,p,o)  = RDFLoad('uri');
out_plain^(s,p,o) = FILTER in_plain^(s,p,o) BY { TP_1 ... TP_N };
⇒
in1_plain^(s,p,o) = RDFLoad('uri');
f1_plain^(s,p,o)  = FILTER in1_plain^(s,p,o) BY { TP_1 };
...
inN_plain^(s,p,o) = RDFLoad('uri');
fN_plain^(s,p,o)  = FILTER inN_plain^(s,p,o) BY { TP_N };
out_plain^(s,p,o) = JOIN f1_plain^(s,p,o) BY (s), ..., fN_plain^(s,p,o) BY (s);
```

Each filter operation can then be further processed by rules (F1) - (F3).

(J2) As already stated in previous sections, the star join can easily be realized using FILTER operators if the input is already in the triple group format. If the triple pattern contains bound variables, e.g. in the predicate position with values $p_1, ..., p_N$, a filter has to be applied.

```
out_group^(s,{(p,o)}) = FILTER in_group^(s,{(p,o)})
                        BY { TP_1 ... TP_N };
⇒
tmp_group^(s,{(p,o)},cnt1,...,cntN) = FOREACH in_group^(s,{(p,o)}) {
  t1 = FILTER stms BY p == 'p_1';
  ...
  tN = FILTER stms BY p == 'p_N';
  GENERATE *, COUNT(t1) AS cnt1, ..., COUNT(tN) as cntN;
};
out_group^(s,{(p,o)},cnt1,...,cntN) = FILTER tmp_group^(s,{(p,o)},cnt1,...,cntN)
                                      BY cnt1 > 0 AND ... AND cntN > 0;
```

(J3) A path join is characterized by the fact that the join variables are not on the same positions in all triple pattern, e.g., the object of TP_i is joined with the subject of TP_{i+1}. If a triple pattern contains a constant that can be used as a filter, those filters can be pushed down.

```
in_plain^(s,p,o)  = RDFLoad('uri');
out_plain^(s,p,o) = FILTER in_plain^(s,p,o) BY { TP_1 ... TP_N };
⇒
in1_plain^(s,p,o)       = RDFLoad('uri');
filter1_plain^(s,p,o)   = FILTER in_plain^(s,p,o) BY { TP_1 };
...
inN_plain^(s,p,o)       = RDFLoad('uri');
filterN_plain^(s,p,o)   = FILTER inN_plain^(s,p,o) BY { TP_N };
out1_plain^(schema_filter1,schema_filter2) = JOIN filter1_plain^(s,p,o) BY (o),
                                             filter2_plain^(s,p,o) BY (s);
...
outi_plain^(schema_outi-1,schema_filteri+1) = JOIN out_i-1_plain^(schema_outi-1) BY (o),
                                              filteri+1_plain^(s,p,o)   BY (s);
...
out_plain^(schema_outN-1,schema_filterN)    = JOIN out_N-1_plain^(schema_outN-1) BY (o),
                                              filterN_plain^(s,p,o)     BY (s);
```

The filter operations on each triple pattern can be processed by rules (F1)-(F3)

(J4) In case the input is loaded as triple groups and a path join needs to be evaluated, we have to flatten the structure and pass the flat triples to the join as in rule (J3) above.

$in_{group}^{(s,\{(p,o)\})}$ = RDFLoad('uri');
$out_{group}^{(s,\{(p,o)\})}$ = FILTER $in_{group}^{(s,\{(p,o)\})}$
BY { TP_1 ... TP_N };
⇒
$in1_{group}^{(s,\{(p,o)\})}$ = RDFLoad('uri');
$filter1_{group}^{(s,\{(p,o)\})}$ = FILTER $in_{group}^{(s,\{(p,o)\})}$ BY { TP_1 };
$flat1_{plain}^{(s,p,o)}$ = FOREACH $filter1_{group}^{(s,\{(p,o)\})}$ GENERATE s, FLATTEN(stmts);
...
$inN_{group}^{(s,\{(p,o)\})}$ = RDFLoad('uri');
$filterN_{group}^{(s,\{(p,o)\})}$ = FILTER $inN_{group}^{(s,\{(p,o)\})}$ BY { TP_N };
$flatN_{plain}^{(s,p,o)}$ = FOREACH $filterN_{group}^{(s,\{(p,o)\})}$ GENERATE s, FLATTEN(stmts);
$out1_{plain}^{(schema_{flat1},schema_{flat2})}$ = JOIN $flat1_{plain}^{(s,p,o)}$ BY (o), $flat2_{plain}^{(s,p,o)}$ BY (s);
...
$outi_{plain}^{(schema_{outi-1},schema_{flati+1})}$ = JOIN $out_{i-1}{}_{plain}^{(schema_{outi-1})}$ BY (o),
$flati+1_{plain}^{(s,p,o)}$ BY (s);
...
$out_{plain}^{(schema_{outN-1},schema_{flatN})}$ = JOIN $out_{N-1}{}_{plain}^{(schema_{outN-1})}$ BY (o),
$flatN_{plain}^{(s,p,o)}$ BY (s);

UDFs and Compiling. We implemented a few UDFs to load and process linked datasets. We use the `SPARQLLoader` that accepts a URL of a SPARQL endpoint and a query to retrieve data from that endpoint. Furthermore, there is a `RDFize` UDF that is used to read and parse statements in N3 files. For compiling and optimizing the resulting script we completely rely on the Pig compiler to generate the MapReduce programs.

6 Evaluation

In this section, we show that the performance of the scripts depends on which rules are applied to transform the extended Pig script into plain Pig. To do so, we compare the execution times for scripts that were generated by different rules. These experiments show that there is a need for an optimizer that has access to various statistics in order to decide which rule to choose.

In our experiments, we used a Hadoop cluster of eight nodes, consisting of one name node and seven data nodes. Each node has an Intel Core i5-3470S CPU with four cores running at 2.9 GHz and 16 GB RAM. The cluster is managed by Cloudera CDH 5.2 running Hadoop 2.5 and Pig 0.12. We imported an 8 GB dataset, which we introduced in Sect. 3, that contains ∼54 million statements.

In the experiments, we manually transformed the extended Pig statements into plain Pig for typical operations on RDF data such as filtering on BGP and self joins. There are

two scenarios for input data: (i) the input data are plain N3 triples (based on which triple groups might be generated on-the-fly) and (ii) the input data is in triple group format. The third case, where data is fetched from a remote SPARQL endpoint, is not considered in this evaluation as the delays caused by remote communication will not lead to additional insights.

For these scenarios, we first compared the execution times of a self join. The task is to compute the latitude and longitude for each event. We started with a Pig script that uses extended Pig features for loading and evaluating a BGP:

```
raw = RDFLoad('hdfs:///eventful.nt')
res = FILTER raw BY  {
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat>  ?lat .
  ?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long }
```

The script includes one `LOAD` and one `FILTER` operator. For `LOAD` we have the two scenarios mentioned above. Depending on the chosen rewriting for `LOAD`, we have two scenarios for the BGP processing: on plain triples or on triple groups. For the latter case, the triple groups are either read directly from the input file or generated on-the-fly. This results in (theoretically) four execution scenarios: (i) data is loaded as plain triples and the filter is evaluated on plain triples, (ii) data is loaded as plain triples and the filter is evaluated on triple groups (this requires generating the groups on-the-fly), (iii) data is loaded as triple groups and evaluation is on plain triples, and (iv) input is loaded as triple groups and filter is evaluated on these groups. We do not consider case (iii) in this evaluation because converting the compact form to the verbose N3 format will obviously not be beneficial. For scenario (iv) the triple groups were generated beforehand using the code shown in Sect. 4. The result was saved to HDFS using Pig's `BinStorage`.

For scenario (i) the above extended Pig script can be rewritten using rule (L1) for the load operation and (J1) for the BGPs:

```
raw1 = RDFFileLoad('hdfs:///eventful.nt');
f1 = FILTER triples1 BY predicate ==
          '<http://www.w3.org/2003/01/geo/wgs84_pos#long>';
raw2 = RDFLoad('hdfs:///eventful.nt');
f2 = FILTER triples2 BY predicate ==
         '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>';
res = JOIN f1 BY (subject), f2 BY (subject);
```

To rewrite the input scenario (ii), we used rules (L1) for the `LOAD` and (T1) to create the triple groups. To rewrite the BGP in the filter expression, rule (J2) was used. For scenario (iv) the rules (L2) and (J2) were applied.

The execution times of the different rewriting results are shown in Fig. 2. For each script, we used one to eight partitions per input dataset. One partition is then processed by one mapper. For scenario (i) this means that because we need to load the dataset twice to perform the join, we actually have twice as many mappers as partitions. For both scenarios (ii) and (iv) we only need to load the dataset once.

As we can see in the figure, working with triple groups is clearly faster than working with plain triples. Although these gaps decrease when increasing the number of partitions,

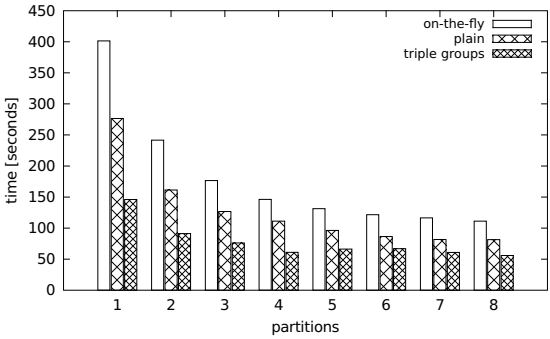


Figure 2: Execution times for self join operation with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

note that for plain triples there are actually twice as many mappers as for the other two scenarios, i.e., this scenario consumes much more resources. However, if the triple groups have to be generated on-the-fly first, there is a significant performance overhead and the processing takes significantly longer as for the scenario where the groups were loaded from a file. As expected, the results suggest that it may be beneficial to materialize such intermediate results, since this would allow other operators in complex scripts to reuse these results. If the materialized results are stored permanently, the optimizer can make use of it in other scripts, too.

Further, we can see the impact of the number of mappers used for execution. When increasing the number of partitions, Pig can make use of more mappers to process the data in parallel, which reduces the overall processing time.

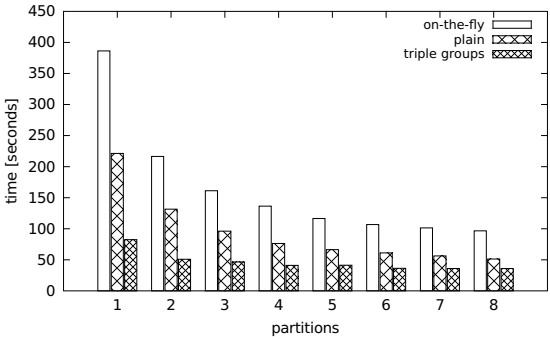


Figure 3: Execution times for a BGP filter on a non-grouping component (object) with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

Next, we measured the performance for a filter on the object component of a triple. The extended Pig script is similar to the one above, despite the filter contains only one triple pattern:

```
raw = RDFLoad('hdfs:///eventful.nt');
res = FILTER raw BY { ?s ?p 'Metallica'. };
```

Similar to the self join script above, we evaluated three rewriting scenarios. For scenario (i), the load operation is also rewritten using rule (L1) and the filer using rule (F2). In scenario (ii), we applied the rules (L1), (T1), and (F5). And in scenario (iv), the rules (L2) and (F5) were used. As an example, we show the rewriting result for scenario (ii):

```
raw = RDFFileLoad('hdfs:///eventful.nt');
out = TUPLIFY(raw);
tmp = FOREACH out {
        t = FILTER stmts BY (object == 'Metallica');
        GENERATE *, COUNT(t) as cnt;
      };
res = FILTER tmp BY cnt > 0;
```

The execution times for the three input scenarios are shown in Fig. 3.

Here, the differences between the scenarios are even greater. Since filtering is not as expensive as a join operation, the overhead of creating the triple groups on-the-fly causes a greater gap in comparison to the other two scenarios. However, we can clearly see that the triple groups (when loaded directly from file) are still faster than filtering plain triples. Since the filter was on the object, the filter operation had to process each tuple in the bag for each subject, which leads to inspecting just as many triples as when evaluation plain triples. In this case, the advantage comes from the fewer data that has to be read from disk, since the subject is stored only once. Hence, when the dataset becomes larger, the differences will increase to the advantage of the triple groups. Also, if the filter has to be evaluated on the component on which the triple groups are grouped, the difference between filtering triple groups and plain triples becomes even greater because the filter predicate has to be evaluated only once per grouped entity (e.g., subject), see Fig. 4. In contrast, for plain triples the filter is evaluated for each statement that belongs to an entity, leading to significantly more filter operations.

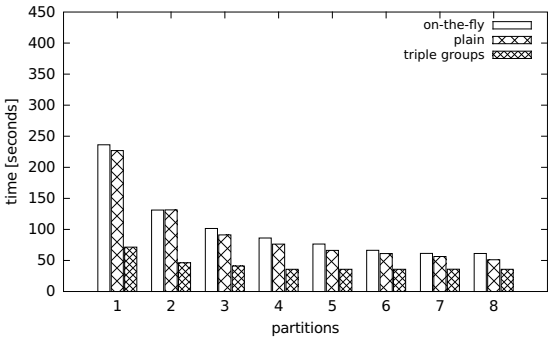


Figure 4: Execution times for a BGP filter on a grouping component (subject) with different numbers of partitions. Left: scenario (ii), middle: scenario (i), and right: scenario (iv).

One can see that the number of mappers also has an impact on the execution time. In our small setup, these differences do not seem to be big enough to be significant. However,

if there is a difference in such a small setup, the difference will be even greater when the number of nodes in the cluster that can participate in the computation or the size of the dataset to process increases.

In this evaluation, we forgo experiments with a path join, because our rule set transforms triple groups to plain triples and then relies on Pig join. Hence, an evaluation of this operation will not reveal new insights.

7 Conclusion

In this paper, we extended the Pig language with support of SPARQL-like features, such as filtering and joins on BGPs, which are particularly useful to perform analytical tasks on linked datasets. The extensions were introduced at three levels of implementation: as strategies for query planning and data access, as rewriting rules to transform the Pig extensions to plain Pig, and as UDFs that implement new functionality. In doing so, we are able to use the Pig compiler to compile and optimize the scripts for execution. The current implementation of our system contains the UDFs and macros to load RDF data either from HDFS or from remote sources via SPARQL endpoints. The implementation is also capable of generating, loading, processing, and storing data in the triple group format.

The results of our experiments show that performance depends on which rules are used for in the rewriting step. To show this effect, the scripts were generated manually using a set of rewriting rules. To support arbitrary queries, we plan to implement a rewriter in our future work that combines the planning and rewriting steps into one optimization step. The optimizer will use heuristics to automatically decide which rules to use for rewriting extended Pig and to decide whether intermediate results should be materialized. To facilitate these decisions, statistics and metadata about the datasets will be obtained from the LODHub platform, which provides the context for our work.

Acknowledgements

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22

References

[ABK+07] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*, 2007.

[ADE+13] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen. Fusion Cubes: Towards Self-Service Business Intelligence. *IJDWM*, 9(2):66–88, 2013.

[BEG+11] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *PVLDB'11*, 2011.

[BL06] T. Berners-Lee. Linked Data. W3C Design Issues. `http://www.w3.org/DesignIssues/LinkedData.html`, 2006.

[CGMR14] D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatis. RDF analytics: lenses over semantic graphs. In *WWW'14*, pages 467–478, 2014.

[DG08] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[EV12] L. Etcheverry and A. A. Vaisman. QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In *COLD'12*, 2012.

[Fou09] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org/core/`, 2009.

[GHS14] L. Galárraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. In *WWW'14*, pages 267–268, 2014.

[HMM+11] M. Husain, J. McGlothlin, M. M Masud, L. Khan, and B. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[HS14] S. Hagedorn and K.-U. Sattler. LODHub - A Platform for Sharing and Analyzing large-scale Linked Open Data. In *Proc. of ISWC 2014 demo track*, volume 1272, pages 181–184, October 2014.

[KM14] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, pages 1–25, 2014.

[KRA11] H. Kim, P. Ravindra, and K. Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. *PVLDB*, 4(12):1426–1429, 2011.

[KUBM12] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig. In *ISWC'12*, pages 247–262. 2012.

[NW08] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD'08*, pages 1099–1110, 2008.

[PKT+13] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *IEEE International Conference on Big Data*, pages 255–263, 2013.

[RKA11] P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC'11*, pages 46–61, 2011.

[SPZL11] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *SWIM'11*, pages 4:1–4:8, 2011.

[W3C04] W3C. RDF Primer (W3C Recommendation 2004-02-10). `http://www.w3.org/TR/rdf-primer/`, 2004.

[W3C08] W3C. SPARQL Query Language for RDF (W3C Recommendation 2008-01-15). `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[W3C10] W3C. Describing Linked Datasets with the VoID Vocabulary. `http://www.w3.org/TR/void/`, 2010.

[W3C13] W3C. The RDF Data Cube Vocabulary. `http://www.w3.org/TR/2013/CR-vocab-data-cube-20130625/`, 2013.

[Wic] Marc Wick. GeoNames. `http://www.geonames.org/`.

[ZCTW13] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE'13*, pages 565–576, 2013.

[ZYW+13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. In *PVLDB'13*, pages 265–276, 2013.

Implementierung von IR-Modellen auf Basis spaltenorientierter Datenbanken oder invertierter Listen

Thomas Stadelmann, Daniel Blank, Andreas Henrich*

Lehrstuhl für Medieninformatik
Universität Bamberg, An der Weberei 5, 96047 Bamberg
thomas.stadelmann@stud.uni-bamberg.de
{daniel.blank|andreas.henrich}@uni-bamberg.de

Abstract: Im Information Retrieval (IR) wird die Anwendung spaltenorientierter Datenbankmanagementsysteme (DBMS) diskutiert, um u.a. durch die Trennung von Datenhaltung und Suchlogik zusätzliche Flexibilität zu gewinnen. Es stellt sich die Frage, ob sich solche Systeme für den praktischen Einsatz eignen, oder ob deren Einsatz auf das Prototyping beschränkt ist. Ziel dieser Arbeit ist es daher, IR-Systeme auf Basis spaltenorientierter DBMS mit konventionellen IR-Bibliotheken auf Basis invertierter Listen bzgl. ihrer Effektivität und Effizienz unter Verwendung des weit verbreiteten Okapi BM25 Retrieval-Modells zu vergleichen. Dabei werden bisherige Arbeiten insbesondere im Hinblick auf die Anzahl und den Typ der untersuchten Anfragen sowie die durchgängige Verwendung von Kompressionsmöglichkeiten erweitert.

1 Einleitung

Für die Implementierung von IR-Systemen wurden bisher hauptsächlich auf deren Einsatzzweck speziell zugeschnittene Datenstrukturen verwendet. Vor allem invertierte Listen haben sich dabei bewährt [MB11]. In letzter Zeit wurden jedoch vermehrt alternative Ansätze für die Umsetzung von IR-Systemen präsentiert. Einen Schwerpunkt bilden dabei IR-Systeme auf Basis spaltenorientierter DBMS, sog. Column Stores. Aufgrund einer sauberen Trennung von Daten und Suchlogik lässt sich durch die Verwendung spaltenorientierter DBMS zusätzliche Flexibilität gewinnen [MSLd14]. So kann das Retrieval-Modell mittels deklarativer Sprachen (z.B. SQL) ohne genauere Kenntnis der Datenrepräsentation sehr einfach angepasst werden. Im praktischen Einsatz stellen neben der Flexibilität allerdings vor allem Effektivitäts- und Effizienzcharakteristika die wesentlichen Aspekte bei der Beurteilung eines IR-Systems dar. Diese Arbeit beleuchtet daher die Frage, inwiefern datenbankbasierte Ansätze die Leistungsfähigkeit konventioneller IR-Systeme im Hinblick auf deren Effektivität und Effizienz erreichen können.

Wir vergleichen fünf frei verfügbare Suchbibliotheken, um jeweils das Okapi BM25 Retrieval-Modell [JWR00] zu unterstützen. Dabei liegt der Fokus auf einer disjunktiven An-

*Unser Dank gilt Isabel Kahre und Sven Peter für ihre Unterstützung im Zuge dieser Arbeit.

fragebearbeitung mit BM25, bei der in einem Ergebnisdokument nicht alle Anfrageterme zwingend vorkommen müssen. Als spaltenorientiertes DBMS wird MonetDB[1] verwendet. Außerdem werden mit Lucene[2] und MG4J[3] zwei in Java geschriebene IR-Bibliotheken, sowie mit Indri[4] und Xapian[5] eine C- bzw. C++-Bibliothek untersucht. Zunächst wird mit Hilfe einer Testkollektion eine Effektivitätsbewertung der Systeme vorgenommen, bevor im Anschluss daran die Systeme im Kontext einer Wikipedia-Suche auf ihre Effizienz hin analysiert werden.

2 Verwandte Arbeiten

Die vorliegende Arbeit motiviert sich maßgeblich durch die Arbeiten von Mühleisen et al. [MSLd14]. Dort wird vorgeschlagen, Dokumentrepräsentationen in einer spaltenorientierten Datenbank zu speichern und Retrieval-Modelle wie BM25 mittels SQL zu formulieren, um IR-Funktionalitäten zu realisieren. Besondere Relevanz in Bezug auf die Anwendung dieses Vorgehens sehen die Autoren im Bereich des Prototyping. So kann durch die Verwendung der deklarativen Sprache SQL von aufwendigerem, imperativem Code für die Realisierung des jeweiligen Retrieval-Modells abgesehen werden. Auf die Anpassung spezieller Datenstrukturen und Algorithmen kann so bei der Implementierung eines neuen Retrieval-Modells verzichtet werden.

Die Machbarkeit des auf einem spaltenorientierten DBMS beruhenden Ansatzes für das IR demonstrieren Mühleisen et al. durch die Verwendung des quelloffenen DBMS MonetDB und des kommerziellen DBMS VectorWise[6]. Auf beiden Systemen wird das konjunktive BM25 Retrieval-Modell mittels SQL-Statements realisiert. Zur Optimierung der konjunktiven BM25 Anfragebearbeitung wird die Vorberechnung der Termfrequenzen für die Terme der einzelnen Dokumente vorgeschlagen, da diese Termfrequenzen statisch sind. Eine weitere Möglichkeit zur Optimierung sehen Mühleisen et al. in der Sortierung der Term-IDs in der Term-Tabelle, um somit die binäre Suche auf den Term-IDs zu ermöglichen. Mühleisen et al. vergleichen die beiden relationalen spaltenorientierten DBMS MonetDB und VectorWise mit den drei quelloffenen Suchmaschinen Lucene, Indri und Terrier. Als Dokumentenkollektion werden ca. 45 Millionen Dokumente einer Web-Kollektion verwendet. Es werden stets 50 Anfragen evaluiert. Dabei wird jeweils zwischen dem ersten Durchlauf (Cold Run) und dem nachfolgenden Durchlauf (Warm Run) unterschieden. Mühleisen et al. kommen in ihren Messungen zu dem Ergebnis, dass die auf spaltenorientierten Datenbanken realisierten IR-Systeme sowohl in der Effektivität als auch in der Effizienz auf einem vergleichbaren Niveau wie konventionelle Ansätze liegen.

Bereits zuvor wurde untersucht, wie sich Retrieval-Modelle mit relationalen DBMS umsetzen lassen (z.B. [GFHR97]). Außerdem existieren Ansätze zur Volltextsuche in vielen

[1] vgl. MonetDB: *https://www.monetdb.org/Home* (letzter Abruf: 12.9.2014)
[2] vgl. Lucene: *http://lucene.apache.org/core/* (letzter Abruf: 12.9.2014)
[3] vgl. MG4J: *http://mg4j.di.unimi.it/* (letzter Abruf: 12.9.2014)
[4] vgl. Indri: *http://www.lemurproject.org/indri.php* (letzter Abruf: 12.9.2014)
[5] vgl. Xapian: *http://xapian.org/* (letzter Abruf: 12.9.2014)
[6] vgl. VectorWise: *http://www.actian.com/* (letzter Abruf: 12.9.2014)

gängigen relationalen DBMS, die meist das Boole'sche Retrieval unterstützen. IR-Systeme auf Basis invertierter Listen werden z.B. in [MB11] verglichen. Ein Alleinstellungsmerkmal der Arbeit von Mühleisen et al. ist jedoch die Verwendung spaltenorientierter DBMS.

Zur genannten Arbeit von Mühleisen et al. unterscheidet sich unser Vergleich in folgenden wichtigen Punkten: Zunächst betrachten wir im Gegensatz zu Mühleisen et al. nicht nur 50 Anfragen bei der Effizienzevaluation, sondern über 15000, wodurch ein wesentlich realistischeres Bild gewährleistet wird. Zum Zweiten betrachten wir über alle Verfahren hinweg den Effekt der komprimierten Indexablage und schließlich legen wir den Schwerpunkt auf ein disjunktives Anfragemodell, bei dem auch Dokumente für eine Anfrage relevant sein können, die nicht alle Anfrageterme enthalten. Eine solche Anfragebearbeitung ist insbesondere für spezielle Anfragen wichtig, bei denen in der Regel nur wenige relevante Dokumente existieren. Die Betrachtung disjunktiver Anfragen erscheint auch deshalb wichtig, weil eine konjunktive Anfrageverarbeitung ein deutlich höheres Optimierungspotential für die Algorithmen bietet. Ferner analysieren wir mit Xapian und MG4J mit dessen zwei Indexlayouts IR-Bibliotheken, die von Mühleisen et al. nicht betrachtet werden.

3 Beschreibungen der eingesetzten Techniken und Suchsysteme

Ebenso wie Mühleisen et al. [MSLd14] unterstellen wir die Verwendung des Okapi BM25 Retrieval-Modells. Dieses lässt sich anhand von Formel (1) beschreiben [JWR00, Roe13]:

$$Sim(Q,D) = \sum_{i=1}^{t} \underbrace{\ln \frac{N - n_i + 0{,}5}{n_i + 0{,}5}}_{\text{IDF-Teil}} \cdot \underbrace{\frac{(k_1+1)tf_{di}}{k_1((1-b)+b\frac{dl}{avdl})+tf_{di}}}_{\text{DTF-Teil}} \cdot \underbrace{\frac{(k_3+1)tf_{qi}}{k_3+tf_{qi}}}_{\text{QTF-Teil}} \tag{1}$$

Die Ähnlichkeit zwischen Anfragerepräsentation Q und Dokumentrepräsentation D wird durch Summenbildung über alle t Anfrageterme ermittelt. Einzelne Summanden setzen sich aus drei Teilen zusammen: dem IDF-Teil (inverse Dokumentfrequenz), dem DTF-Teil (Dokumenttermfrequenz) sowie dem QTF-Teil (Anfragetermfrequenz).

Der IDF-Teil modifiziert geringfügig die inverse Dokumentfrequenz $\frac{N}{n_i}$, bei der N für die Gesamtzahl der Dokumente im Korpus steht und n_i die Dokumentfrequenz (*df*) von Term i erfasst. Durch die Verwendung des Logarithmus können negative Werte resultieren. Die von uns evaluierten IR-Systeme gehen auf unterschiedliche Art damit um.

Der DTF-Teil wird maßgeblich von der Termfrequenz tf eines Terms i in einem Dokument d beeinflusst; dl bezeichnet die Länge eines Dokuments (entspricht i.d.R. der Anzahl der Terme), $avdl$ die durchschnittliche Dokumentlänge in der Kollektion. Die beiden Parameter b und k_1 können i.d.R. vom Anwender der IR-Bibliothek konfiguriert werden. Standardwerte sind entsprechend als Vorschläge vorgegeben. Auch hierbei unterscheiden sich die verschiedenen Systeme.

	Version	b	k_1	k_3	MAP	P@10	P@20
Indri	5.6	0,75	1,2	7	0,1904	0,3830	0,3184
Lucene	4.7.2	0,75	1,2	0	0,1837	0,3726	0,3198
MG4J	5.2.1	0,75*	1,2	0	0,1904	0,3802	0,3156
MonetDB	Jan2014-SP2	0,75	1,2	0	0,1904	0,3783	0,3142
Xapian	1.2.17	0,5	1	1	0,1897	0,3783	0,3170

Tabelle 1: Links: Okapi BM25 Parameter der evaluierten Suchsysteme; * zeigt an, dass $b = 0{,}5$ verändert und an den Wert von Lucene angeglichen wurde. Rechts: MAP, P@10 und P@20 bei disjunktivem BM25 auf der Ohsumed-Kollektion.

	docid	*name*	*dl*
docs:	1	Doc1	234
	..	..	..

	termid	*docid*	*tf*
terms:	1	1	2
	..	..	..

	termid	*term*	*df*
dict:	1	hamburg	23
	..	..	..

Abbildung 1: Schema des datenbankbasierten IR-Systems

Der QTF-Teil betrachtet die Vorkommenshäufigkeit eines Anfrageterms tf_{qi}. Da bei vielen Systemen initial $k_3 = 0$ gilt, ergibt sich für den QTF-Teil in diesen Fällen ein Wert von 1, sodass auf eine Anfragetermgewichtung bei Berechnung der Ähnlichkeit verzichtet wird.

Im Folgenden beschreiben wir kurz die wesentlichen Charakteristika und Unterschiede der analysierten IR-Systeme. Diese unterscheiden sich u.a. bei der BM25 Parametrisierung sowie bei den zugrunde liegenden Datenstrukturen. Die voreingestellten BM25-Parameter der Programme sowie die eingesetzte Programmversion fasst Tabelle 1 zusammen.

Datenbankbasierte Lösungen mit MonetDB
Das verwendete Datenbankschema zur Umsetzung des BM25 Retrieval-Modells ist in Abb. 1 dargestellt. Dabei handelt es sich um ein gemäß den verbalen Erläuterungen von [MSLd14] optimiertes Modell mit vorberechneten Termfrequenzen (*tf*). Der SQL-Code aus [MSLd14] für konjunktives BM25 wird von uns übernommen und für das disjunktive Modell entsprechend angepasst (vgl. Abb. 2). Wir verwenden die in Tab. 1 angegebenen Parameter. Außerdem operiert der SQL-Code bei Mühleisen et al. schon auf *termid*-Ebene und nicht, den Anfragen entsprechend, auf *term*-Ebene. Diese Auflösung muss zusätzlich durchgeführt werden. Wir beschränken uns hier auf zwei Varianten. Variante a) legt für jede Abfrage jeweils pro *term* einen SQL-Parameter fest. Dieser wird vor dem eigentlichen SQL-Statement durch ein zusätzliches SQL-Statement mit der *termid* befüllt und in der Abfrage verwendet. Variante b) lädt die komplette *term-termid* Zuordnung (vgl. die rechte Tabelle in Abb. 1) a priori in den Hauptspeicher.

Lösungen auf Basis invertierter Listen
Lucene ist eine häufig verwendete IR-Bibliothek, die seit Version 4 das BM25 Modell unterstützt. In der von uns verwendeten Version 4.7.2 sind die Parameter des BM25 Modells wie folgt gesetzt: $b = 0{,}75$, $k_1 = 1{,}2$ und $k_3 = 0$. Lucene verzichtet demnach auf den QTF-Teil des BM25 Retrieval-Modells und damit auf die Berücksichtigung der Anfragetermfrequenzen. Zudem vermeidet Lucene negative IDF-Teile, indem $ln(\frac{N-n_i+0{,}5}{n_i+0{,}5}+1)$ verwendet wird. Eine weitere Besonderheit ist die Speicherung der Dokumentlänge mit verminderter Präzision. Die Dokumentlänge wird in eine 1 Byte große Gleitkommazahl mit 3 Bit Mantisse transformiert.

```
WITH subscores AS(
  SELECT terms.docid, terms.termid,
          (log((@N-df+0.5)/(df+0.5))*((tf*(1.2+1)/(tf+1.2*(1-0.75+0.75*(dl/@avdl)))))) AS subscore
  FROM terms
  JOIN docs ON terms.docid=docs.docid
  JOIN dict ON terms.termid=dict.termid
  WHERE terms.termid IN (/*list of term IDs*/)
)
SELECT name, score FROM(
  SELECT docid, sum(subscore) AS score
  FROM subscores GROUP BY docid
) AS scores
JOIN docs ON scores.docid=docs.docid ORDER BY score DESC LIMIT 1000;
```

Abbildung 2: SQL-Code für die Bearbeitung disjunktiver Anfragen mittels Variante b).

MG4J verwenden wir mit den gleichen BM25 Parametern wie Lucene. Diese Anpassung erfolgt, um bei der Effektivitätsevaluierung den Einfluss der reduzierten Genauigkeit bei Speicherung der Dokumentlänge in Verbindung mit dem unterschiedlichen Umgang mit negativen IDF-Teilen quantifizieren zu können. Resultiert ein IDF-Teil mit einem Wert kleiner als der Schwellwert 10^{-6}, wird der Schwellwert in den weiteren Berechnungen verwendet. Bei MG4J wird zudem zwischen dem ab Version 5 neu eingeführten Quasi-Succinct-Indexformat [Vig13] und dem traditionellen Format unterschieden.

Indri ist ein IR-System, das von der University of Massachusetts entwickelt wird und häufig im Zusammenhang mit Sprachmodellen Anwendung findet. Es unterstützt jedoch auch eine Anfragebearbeitung mit BM25. Die beiden Parameter b und k_1 bleiben gegenüber Lucene unverändert. Allerdings unterscheidet sich Indri bei Auslieferung mit einem $k_3 = 7$ von den anderen Systemen, wobei die Anfrageterme in den in Abschnitt 4 bei der Evaluierung betrachteten Fällen in der Regel ohnehin jeweils nur 1-mal vorkommen, sodass die QTF-Komponente ohne Einfluss bleibt.

Xapian ist hinsichtlich der BM25 Konfiguration in mehrerlei Hinsicht speziell – zum einen bei der Parameterfestlegung mit $k_1 = 1$, $k_3 = 1$ bzw. $b = 0{,}5$ und zum anderen im IDF-Teil. Sofern $\frac{N-n_i+0{,}5}{n_i+0{,}5} < 2$, wird $ln(\frac{N-n_i+0{,}5}{n_i+0{,}5} \cdot 0{,}5 + 1)$ verwendet.

4 Evaluierung

In unseren Experimenten verwenden wir die in Abschnitt 3 genannten Systeme. Das kommerzielle VectorWise wird nicht evaluiert. Auf eine Evaluation von Terrier verzichten wir ebenso, weil dieses System in [MSLd14] schlechtere Effizienzwerte als bspw. Lucene und Indri aufweist. Wir evaluieren aus diversen Gründen vorwiegend das disjunktive BM25-Modell, u.a. aufgrund der verwendeten Testkollektion, die sonst viele Anfragen mit leerer Ergebnismenge liefern würde, sowie der Eingeschränktheit der Konsolenanwendung von Indri auf diese Art der Anfragebearbeitung. Ferner ist eine disjunktive Anfragebearbeitung insbesondere für spezielle Anfragen mit wenigen relevanten Dokumenten wichtig und in der Bearbeitung in der Regel aufwändiger als die konjunktive Verarbeitung.

Messungen werden so nah wie möglich an der jeweiligen IR-Bibliothek durchgeführt, um mögliche Störungen durch weitere Softwareschichten zu vermeiden. Dies bedeutet, dass für die Java-Bibliotheken jeweils ein Java-Programm geschrieben wurde. Die C-bzw. C++-Bibliotheken werden über deren mitgelieferte Kommandozeilenprogramme direkt mittels Shell-Skript ausgeführt. Datenbanken werden mittels JDBC aus Java angesprochen.

Als Testsystem wird ein Rechner mit Intel Core-i7 860 Vierkern-CPU, 4GB Hauptspeicher und einer 250GB HDD mit 7200 rpm verwendet. Die Festplatte wurde in eine ext4-Partition und eine btrfs-Partition mit aktivierter zlib Kompression aufgeteilt. Dadurch können Messungen sowohl mit als auch ohne Kompression durchgeführt werden. Anders als in [MSLd14] beschränken wir uns bei der Kompression nicht nur auf die spaltenorientierten Datenbanken. Auch die konventionellen IR-Bibliotheken werden mit und ohne Kompression getestet. Als Betriebssystem wird Ubuntu Server 14.04 LTS verwendet. Neben dem OpenJDK wurden auf dem System lediglich die für das Kompilieren der Kommandozeilenanwendungen benötigten Pakete installiert.

Effektivitätsevaluierung auf Basis der Ohsumed-Kollektion
Für die Effektivitätsevaluierung wird die Ohsumed-Kollektion[7] verwendet. Sie besteht aus einer Untermenge der bibliografischen Angaben zu medizinischer Fachliteratur, die in der MEDLINE-Datenbank des US-amerikanischen *National Center for Biotechnology Information* vorgehalten werden. Diese besteht aus 348566 Referenzen aus 270 medizinischen Fachzeitschriften der Jahre 1987 bis 1991. Jede Referenz beinhaltet eine textuelle Kurzbeschreibung, die wir indexieren. Des Weiteren wurden von Ärzten 106 Anfragen an die Testkollektion formuliert. Von einer anderen Ärztegruppe wurden jeweils Dokumente als definitiv und möglicherweise relevant auf eine Anfrage eingestuft. Da die von uns verwendeten Effektivitätsmaße eine binäre Definition von Relevanz erwarten, werden auch die möglicherweise relevanten Dokumente als relevant eingestuft.

Zur weiteren Verarbeitung werden die Kurzbeschreibungen und Anfragen mittels der in der Programmbibliothek von Lucene enthaltenen Werkzeuge vorverarbeitet, sodass alle verwendeten Bibliotheken diesbezüglich einheitlich arbeiten. Die Vorverarbeitung besteht aus Tokenisierung, Entfernung von Stoppwörtern und einem Porter Stemming. Die Tokenisierung kann bei den Kommandozeilensystemen nicht beeinflusst werden, sodass wir diese den jeweiligen Systemen überlassen. Beim datenbankgetriebenen Ansatz kommt die Default-Tokenisierung von Lucene zum Einsatz.

Es werden stets für jedes System die jeweils ersten 1000 Ergebnisse zu einer Anfrage angefordert. Tabelle 1 in Abschnitt 3 zeigt die Effektivitätsmaße Mean Average Precision (MAP), P@10 und P@20. Die insgesamt sehr geringen Abweichungen zwischen den einzelnen Suchtechnologien resultieren vor allem aus einer unterschiedlichen Parametrisierung (vgl. Tabelle 1) und einer unterschiedlichen Umsetzung der IDF-Komponente des BM25 Modells. Zudem trägt bei Lucene die bereits erwähnte verringerte Präzision bei Speicherung der Dokumentlänge zu Effektivitätsverlusten – gemessen mit MAP und P@10 – bei. Interessanterweise werden diese zumindest im Falle der Ohsumed-Kollektion bei P@20 kompensiert.

[7] *http://ir.ohsu.edu/ohsumed/ohsumed.html* (letzter Abruf: 15.9.2014)

	Indri	Lucene	MG4J.S	MG4J.I	MonetDB	Xapian
Indexierungszeit	4:31 Std	6:06 Std	5:01 Std	4:56 Std	7:07 Std	5:13 Std
Indexgröße	4,2 GB	2,8 GB	0,9 GB	1,1 GB	2,9 GB	7,8 GB

Tabelle 2: Indexierungszeiten und unkomprimierte Indexgrößen in Verbindung mit der Wikipedia-Kollektion. Komprimierte Größen lassen sich im btrfs-Dateisystem nicht exakt bestimmen.

Effizienzevaluierung auf Basis der Wikipedia

Da zur Messung der Effizienz keine Relevanzurteile benötigt werden, haben wir uns hier für einen größeren Korpus mit mehr Anfragen entschieden. Die Anfragen stammen aus dem bekannten AOL Query Log[8]. Indexiert wird die englische Wikipedia. Da die Anfragen aus einem Query Log des Jahres 2006 extrahiert werden, wählen wir einen Wikipedia Dump[9] aus dem November 2006. Tabellen und Bildreferenzen werden aus den Dokumenten entfernt. Dadurch entstehende zu kurze Dokumente ohne jeglichen Informationsgehalt werden mittels Schwellenwert entfernt. Es werden nur die Dokumente verwendet, die mindestens acht Wörter enthalten. Als Resultat erhalten wir 1490557 Dokumente. Von den Anfragen des AOL Query Logs werden nur diejenigen verwendet, bei denen unmittelbar nach der Suche auf einen Link der englischen Wikipedia geklickt wurde. Somit passen die Anfragen sowohl inhaltlich als auch zeitlich zur Dokumentkollektion. Ferner wurden die Suchanfragen durch einfache Heuristiken weiter bereinigt (mindestens drei Zeichen, mindestens einen Vokal, keine Internetadresse). Weiter wurden Anfragen vernachlässigt, die Wikipedia als Begriff enthalten. Insgesamt konnten so 15284 Anfragen extrahiert werden.

Während bei der Ohsumed-Kollektion die Entfernung von Stoppwörtern und eine Stammformreduktion von Vorteil ist, um die Effektivität zu steigern, werden solche Verfahren bei Webkollektionen wie unserem Wikipedia Dump kontrovers diskutiert [BR11]. Da Websuchmaschinen durchaus andere Ergebnisse auf unterschiedliche Anfragen des gleichen Wortstammes liefern und das *to-be-or-not-to-be*-Problem [BR11, *p*. 226] gegen eine Stoppwortentfernung spricht, verzichten wir hier auf diese Vorverarbeitungsmaßnahmen. Für die Tokenisierung verwenden wir die gleichen Mechanismen wie zuvor. Tabelle 2 gibt Aufschluss über die Indexierungszeiten und Indexgrößen.

Wir verwenden zunächst das disjunktive BM25 Retrieval-Modell, das bei nahezu allen Anfragen ein Ergebnis liefert (bei 99,7% bzw. 97,5% mit bzw. ohne Verwendung einer Stammformreduktion und Stoppworteliminierung). Bei konjunktivem Modell liefern nur noch 46,2% der Anfragen ohne Stammformreduktion und Stoppworteliminierung ein Ergebnis (mit Stammformreduktion und Stoppworteliminierung sind dies 92,0%).

Bei den Anfragedurchläufen findet wie bei Mühleisen et al. nach dem Cold Run unmittelbar folgend ein Warm Run statt. Vor jedem Cold Run werden Dateisystemcaches geleert. Die Anfragen werden immer in der gleichen Reihenfolge an das jeweilige System gestellt.

Statistiken über die Anfragezeiten aller 15284 Anfragen sind in Tabelle 3 dargestellt. Abbildung 3 (links) zeigt den Warm Run, der sich in der Interpretation der Ergebnisse und Beurteilung der Leistungsfähigkeit der einzelnen Systeme nur bei den Ausreißern vom

[8] *http://www.gregsadetsky.com/aol-data/* (letzter Abruf: 13.10.2014)

[9] *https://dumps.wikimedia.org/archive/2006/2006-12/enwiki/20061130/enwiki-20061130-pages-articles.xml.bz2* (letzter Abruf: 15.9.2014)

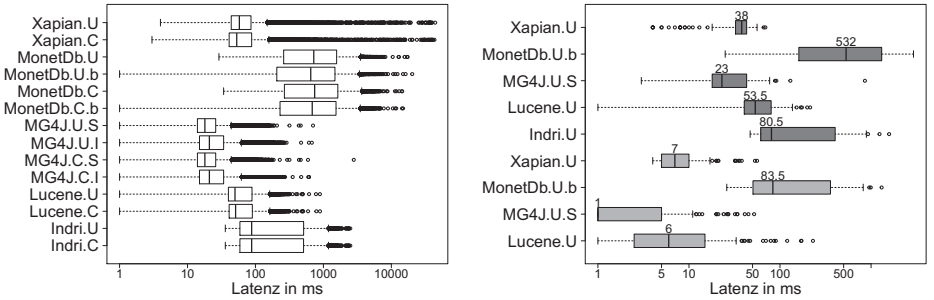


Abbildung 3: Links: Durchläufe über alle 15284 Anfragen für den Warm Run; b steht für Variante b) gemäß Abschnitt 3; U für das ext4 Dateisystem, C für das btrfs Dateisystem mit aktivierter zlib Kompression, I für das alte Indexformat von MG4J, S für das neue Indexformat von MG4J. Rechts: Disjunktive (dunkelgrau) und konjunktive Durchläufe (hellgrau) über 100 Anfragen im Warm Run.

Cold Run unterscheidet, wie auch aus Tab. 3 ersichtlich ist. Dabei ist eine klare Tendenz zu erkennen: MG4J und Lucene stellen die effizientesten Systeme dar. MG4J ist in den meisten Fällen doppelt so schnell wie Lucene. MG4J kann sich mit dem neuen Quasi-Succinct-Indexformat weiter gegenüber Lucene absetzen. Xapian und Indri folgen. Während Xapian im Median noch nahe bei Lucene liegt, zeigen die arithmetischen Mittel und die Boxplots[10], dass Xapian unter Ausreißern leidet. Indri ist bei Betrachtung des Medians und des arithmetischen Mittels darüber hinaus noch etwas langsamer. MonetDB stellt mit Abstand das schlechteste System dar. Mit ca. 650 ms im Median ist das System mehr als 10-mal langsamer als Lucene und gar mehr als 30-mal langsamer als das beste System MG4J mit neuem Indexformat. Zudem verhält sich das MonetDB-System im Gegensatz zu den anderen Systemen so, dass bei der großen Zahl von Anfragen bei der wiederholten Durchführung (Warm Run) sogar geringfügige Effizienzverschlechterungen zu verzeichnen sind. Eine Analyse der Messergebnisse zeigt, dass die Anfragezeiten bei den Ausreißern ansteigen. Bei Mühleisen et al. führt ein Warm Run mit lediglich 50 Anfragen zu einer Beschleunigung der Laufzeit. Für eine geringe Anzahl an 100 Anfragen ist dies bei uns ebenso der Fall (sowohl bei konjunktiver als auch disjunktiver Anfragebearbeitung). Allerdings verschlechtert sich das Laufzeitverhalten des Warm Runs gegenüber des Cold Runs, sofern 15284 Anfragen untersucht werden. Durchweg positiv stellt sich Variante b) dar und damit das Halten der *term-termid* Zuordnung im Hauptspeicher. Damit arbeitet das System ca. 60 ms schneller.

Vergleicht man die Ergebnisse auf komprimierten und unkomprimierten Dateisystemen ergeben sich weitere Auffälligkeiten. Während alle Systeme außer MonetDB von der Kompression im Cold Run profitieren, sind beim Warm Run keine positiven Auswirkungen mehr sichtbar. Es zeigt sich, dass durch die Kompression vor allem Ausreißer vermieden werden. Durch die Tatsache, dass die Systeme zunächst Teile des Index von der Fest-

[10] Sofern keine Ausreißer eingezeichnet sind, handelt es sich bei den Whiskern um das Minimum bzw. Maximum; andernfalls um das 1,5-fache des Interquartilabstands.

	Median				arithmetisches Mittel			
Kompression	inaktiv		aktiv		inaktiv		aktiv	
Durchlauf	cold	warm	cold	warm	cold	warm	cold	warm
MG4J.S	22	18	18	18	31	23	27	23
MG4J.I	25	21	22	18	35	29	33	29
Lucene	52	50	50	51	75	64	63	64
Xapian	62	58	56	53	246	188	230	195
Indri	102	88	98	88	293	277	285	278
MonetDB.b	643	652	657.5	685	884	937	914	963
MonetDB	705	722	717	744	942	1009	973	1016

Tabelle 3: Median und arithmetisches Mittel in Millisekunden der Durchläufe bei allen 15284 Anfragen auf ext4 Dateisystem und btrfs Dateisystem mit aktiver zlib Kompression; b steht für Variante b) gemäß Abschnitt 3; S für das neue MG4J-Indexformat, I für das alte MG4J-Indexformat.

platte in den Hauptspeicher laden müssen, scheint es schlüssig, dass Ausreißer vor allem während des Startvorgangs auftreten.

Dieses Verhalten ist für Lucene am ausgeprägtesten. Ein ähnliches Verhalten ist bei den anderen konventionellen IR-Bibliotheken ebenfalls feststellbar. Lucene erreicht erst nach ca. 100 Anfragen ein konstantes Latenzniveau. Durch Dateisystemkompression kann der Start des Systems abgesehen von der ersten Anfrage stark beschleunigt werden. Die Evaluierung über alle 15284 Anfragen zeigt jedoch, dass sich die Kompression nach den ersten 100 Anfragen kaum noch positiv auswirkt.

Der rechte Plot in Abb. 3 stellt eine konjunktive und disjunktive Anfragebearbeitung bei 100 zufällig gewählten Anfragen im Warm Run gegenüber und verdeutlicht, dass die disjunktive Anfragebearbeitung bei allen Systemen deutlich mehr Aufwand verursacht als die konjunktive Verarbeitung. Während bei der disjunktiven Variante Lucene schlechter als Xapian abschneidet, ist diese Tendenz im linken Teil der Abbildung bei Betrachtung aller Anfragen nicht klar vorhanden. Abb. 3 (rechts) bestätigt außerdem für disjunktive Anfragen die bereits für konjunktive Anfragen getroffenen Aussagen von Mühleisen et al., die zeigen, dass bei Betrachtung weniger Anfragen im Warm Run Lucene besser als Indri arbeitet, letzteres auch besser als MonetDB. Die Abbildung zeigt auch, dass sowohl bei konjunktiver als auch bei disjunktiver Anfragebearbeitung MG4J das leistungsfähigste System ist, gefolgt von Xapian. Bei Betrachtung aller 15284 Anfragen und disjunktiver Anfragebearbeitung (vgl. Abb. 3, links) ist Xapian jedoch Lucene unterlegen.

Aus Platzgründen nicht aufgeführte Laufzeitmessungen auf Basis der Ohsumed-Kollektion, die sich durch mehr Terme pro Anfrage (5,44 in Relation zu 3,87) auszeichnet, bestätigen zudem die Überlegenheit der konventionellen IR-Systeme gegenüber MonetDB.

5 Fazit

In dieser Arbeit haben wir ein auf dem spaltenorientierten DBMS MonetDB basierendes IR-System mit vier verschiedenen konventionellen IR-Systemen verglichen. Dabei wurde als Retrieval-Modell Okapi BM25 verwendet. Als Dokumentkollektion wurde die Wiki-

pedia aus dem Jahr 2006 verwendet, sowie reale Anfragen des AOL Query Logs aus dem gleichen Jahr. Wir konnten zeigen, dass hinsichtlich der Effizienz bei der Anfragebearbeitung ein deutlicher Unterschied zwischen den performantesten IR-Bibliotheken und einem auf dem spaltenorientierten DBMS MonetDB basierenden IR-System existiert. Lucene ist etwa 10-mal so schnell wie das MonetDB-System, MG4J mit neuem Indexformat gar 30-mal schneller. Das schlechteste konventionelle System ist immer noch über 6-mal schneller als das MonetDB-System. Damit eignet sich das MonetDB-System gut für IR-Prototypen und Effektivitätsevaluierungen in moderatem Umfang. Für einen produktiven Einsatz wie etwa als Websuchmaschine oder zur Unternehmenssuche sind IR-Systeme auf Basis invertierter Listen jedoch besser geeignet.

Des Weiteren haben wir das Startverhalten von konventionellen IR-Bibliotheken analysiert. Dabei konnten wir feststellen, dass diese beim Start einige Anfragen benötigen, um ein performantes Anfragezeitniveau zu erreichen. Besonders auffällig ist dieses Verhalten bei Lucene. Dort werden ca. 100 Anfragen benötigt bis das System warm ist. Durch Dateisystemkompression kann der Startvorgang stark beschleunigt werden. Im weiteren Verlauf wirkt sich die Kompression allerdings nicht mehr positiv auf die meisten Systeme aus. Mit dieser Erkenntnis können die Messungen von [MSLd14] in deren ersten Lauf im Speziellen für Lucene relativiert werden. Denn dort wurden lediglich 50 Anfragen an das System gestellt, wodurch das System sich noch deutlich in der Startphase befand. Wir konnten zudem zeigen, dass Lucene und andere konventionelle IR-Bibliotheken von Caching-Mechanismen profitieren und unabhängig von der konkreten Anfrage anfangs stetig an Performanz gewinnen.

Literatur

[BR11] Ricardo A. Baeza-Yates und Berthier A. Ribeiro-Neto. *Modern Information Retrieval – the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[GFHR97] David A. Grossman, Ophir Frieder, David O. Holmes und David C. Roberts. Integrating structured data and text: A relational approach. *Journal of the American Society for Information Science*, 48(2):122–132, 1997.

[JWR00] K. Sparck Jones, S. Walker und S. E. Robertson. A Probabilistic Model of Information Retrieval: Development and Comparative Experiments. *Inf. Process. Manage.*, 36(6):779–808, November 2000.

[MB11] Christian Middleton und Ricardo A. Baeza-Yates. Open Source Search Engines. In Ricardo A. Baeza-Yates und Berthier A. Ribeiro-Neto, Hrsg., *Modern Information Retrieval – the concepts and technology behind search, Second edition*, Kapitel A. Pearson Education Ltd., Harlow, England, 2011.

[MSLd14] Hannes Mühleisen, Thaer Samar, Jimmy Lin und Arjen de Vries. Old Dogs Are Great at New Tricks: Column Stores for IR Prototyping. In *Proc. of the 37th Intl. SIGIR Conf. on Research and Development in Information Retrieval*, Seiten 863–866, Gold Coast, Queensland, Australia, 2014. ACM.

[Roe13] Thomas Roelleke. Information Retrieval Models: Foundations and Relationships. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 5(3):1–163, 2013.

[Vig13] Sebastiano Vigna. Quasi-succinct Indices. In *Proc. of the 6th Intl. Conf. on Web Search and Data Mining*, Seiten 83–92, Rom, Italien, 2013. ACM.

Benchmarking

Hochperformante Analysen in Graph-Datenbanken

Moritz Kaufmann, Tobias Mühlbauer, Manuel Then,
Andrey Gubichev, Alfons Kemper, Thomas Neumann
Technische Universität München
{kaufmanm,muehlbau,then,gubichev,kemper,neumann}@in.tum.de

Abstract: Ziel des ACM SIGMOD Programming Contest 2014 war es ein hochperformantes System für die Analyse von großen Graph-Daten zu entwickeln. Insbesondere die unregelmäßigen Speicherzugriffsmuster und Kontrollflussverzweigungen von Graphalgorithmen stellen dabei eine große Herausforderung dar, da diese bisher nicht effizient auf modernden superskalaren Mehrkern-Prozessoren ausgeführt werden können. Um diese Prozessoren optimal auszulasten bedarf es zudem der Nutzung aller parallelen Ausführungseinheiten.

In der vorliegenden Arbeit präsentieren wir das Gewinnersystem des Wettbewerbs. Der Erfolg unseres Systems beruht, neben gutem Engineering, auf den folgenden Entwicklungen: (i) Daten-parallelisierte Graph-Breitensuche, welche Cache-Misses effizient amortisiert, (ii) Heuristiken zur Reduzierung des Suchraums bei Top-k-Anfragen, (iii) schnelles parallelisiertes Laden von textuellen Rohdaten, und (iv) feingranulares Task-Scheduling um Mehrkern-Prozessoren optimal auszulasten. Die in dieser Arbeit beschriebenen Neuentwicklungen werden derzeit in unser Hauptspeicher-Datenbanksystem HyPer integriert und lassen sich unserer Einschätzung nach auch in bestehende Graph-Datenbanksysteme integrieren.

1 Einführung

Mit Facebook und Google+ existieren heute soziale Netzwerke mit mehreren Hundert Millionen aktiven Benutzern. Die Beziehungen der Menschen, ihre Interessen und Beiträge ergeben dabei einen Graphen mit Millionen von Knoten, Kanten und Eigenschaften. Neben sozialen Graphen existieren Graph-Daten unter anderem auch für Straßen- und Schienennetze und die Verlinkung von Webseiten. Durch die Analyse dieser Graph-Daten können nicht nur werberelevante Zielgruppen bestimmt werden, sondern auch wichtige Erkenntnisse für die Sozialwissenschaften, Medizin und Astronomie gewonnen werden.

Graphen lassen sich in traditionellen relationalen Datenbanken abbilden, indem Knoten und Kanten in Relationen gespeichert werden. Auch können Graphalgorithmen in SQL-Anfragen übersetzt werden. Zur Traversierung der Graph-Daten werden dazu häufig rekursive SQL-Anfragen (mit sog. *Recursive Common Table Expressions*) benutzt. Es ist jedoch leicht ersichtlich, dass die rekursive Berechnung von Joins sehr schnell zu einer hohen Komplexität bei der Berechnung der Anfrageergebnisse führt. Um diesem Engpass bei der Analyse großer Graph-Daten zu begegnen, werden seit einigen Jahren spezialisierte Graph-Datenbanksysteme in Industrie und Forschung entwickelt. Diese zeichnen sich durch eingebaute Graphalgorithmik und eine für diese Algorithmik angepass-

te Speicherung der Daten aus. Die mitgelieferten Algorithmen erlauben unter anderem die Traversierung von Graphen durch Tiefen- und Breitensuche und das Auffinden von kürzesten Pfaden und Cliquen. Zu den bekanntesten Graph-Datenbanksystemen gehören Neo4j [Neo14], HyperGraphDB [Ior10], und Sparksee (früher DEX) [MBMMGV+07]. Mit Pregel [MAB+10], FlockDB [Flo14] und Trinity [SWL13] haben Google, Twitter und Microsoft zudem jeweils ein Framework zur Analyse von großen Graph-Datenbanken in verteilten Systemen entwickelt. Neueste Forschungsergebnisse zeigen zudem, dass auch spaltenorientierte relationale Datenbanksysteme [JRW+14] zu den zuvor genannten Systemen konkurrenzfähig sein können.

Der ACM SIGMOD Programming Contest setzt sich zum Ziel jedes Jahr die Aufmerksamkeit auf eine forschungsrelevante Aufgabenstellung zu lenken und Studenten und Doktoranden zu animieren in einem Zeitraum von drei Monaten nach möglichst effizienten Lösungen zu suchen. Ziel des diesjährigen, mit einem Preisgeld von 5.000 US-$ dotierten, Programmierwettbewerbs war es die technischen Möglichkeiten eines Systems zur Analyse großer sozialer Graphen auf einem modernen superskalaren Mehrkern-Prozessor zu ermitteln. Die konkrete Aufgabe bestand darin vier verschiedene, parametrisierte Anfragetypen auf einem synthetisch generierten sozialen Graphen möglichst schnell zu beantworten [Pcs14]. Der Datengenerator wurde dabei dem *The Social Network Benchmark* des Linked Data Benchmark Council (LDBC) [BFG+13] entnommen[1]. Neben der Korrektheit war auch die Skalierbarkeit der Lösung bis hin zu sehr großen Graphen mit mehreren Millionen Knoten gefordert. Bezüglich der Ausführung waren die Programme des Wettbewerbs auf eine nicht verteile Umgebung beschränkt. Mit heutiger Rechenleistung und Arbeitsspeichermengen jenseits eines Terabytes in einem Server ist es aber möglich selbst Graphen mit mehreren hundert Millionen Knoten und Kanten im Hauptspeicher auf einem einzigen Server zu analysieren [APPB10]. Die effiziente Ausführung von Graphalgorithmen auf solchen Servern stellt allerdings eine große Herausforderung dar [LGHB07]. Die meisten Algorithmen zur Graphanalyse besitzen unregelmäßige Speicherzugriffsmuster und eine Vielzahl an Kontrollflussverzweigungen. Dies wird besonders bei Graph-Daten, die größer als die Caches auf modernen Prozessoren sind zum Problem, da die Ausführungszeit dann durch die Speicherzugriffslatenz bestimmt wird. Es ist daher unerlässlich neue Daten-parallele Algorithmen zu entwickeln, welche die Speicherzugriffslatenzen, die durch Cache-Misses entstehen, effizient amortisieren. Zudem ist es auch hilfreich, die Anzahl der schwer vorhersagbaren Kontrollflussverzweigungen und die damit verbundene Anzahl der *Branch-Misses* zu minimieren. Um moderne superskalare Mehrkern-Prozessoren optimal auszulasten, ist neben der Nutzung von Daten-Parallelität auch die parallele Ausführung auf allen Prozessorkernen notwendig. Dies erschwert die Implementierung von Graphalgorithmen zunehmend, da die Ausführungsgeschwindigkeit in vielen Dimensionen beeinflusst wird.

Durch die Implementierung von effizienten Graphalgorithmen, welche für Server mit modernen Mehrkern-Prozessoren und einem großen Hauptspeicher optimiert sind, war es mehreren Teams des SIGMOD Programming Contest möglich die Anfragezeiten der im Wettbewerb gestellten Aufgabentypen um mehr als eine Größenordnung gegenüber bisher

[1] Das LDBC [BFG+13] ist ein EU-Projekt, welches sich die Entwicklung von Benchmarks für die Analyse großer Graph- und RDF-Datenbanken zum Ziel gesetzt hat.

Rank	Team	Small (215 MB) [s]	Medium (7 GB) [s]	Large (23 GB) [s]
1	**AWFY (TUM)**	0.126	0.237	2.849
2	~~unknown~~	0.124	0.331	3.674
3	VIDA (NYU)	0.207	0.469	4.571
4	H_minor_free (UToyko)	0.127	0.366	6.046
5	~~VSB_TUO~~	0.185	0.699	6.824
6	~~Bolzano_Nguyen~~	0.127	0.497	9.833
7	blxlrsmb (Tsinghua Univ.)	0.126	0.555	9.852
8	Cracker (Peking Univ.)	0.193	0.765	12.276
9	UCY_YouSeeWhy	0.228	1.193	14.255
10	GGWP	0.471	2.372	28.689
11	Cloud11	1.358	2.651	29.411
12	zero	0.564	3.097	35.459
13	VedgeX	2.775	3.629	36.537
14	LogisticAggression	0.310	2.510	54.592
15	QQn00bs	0.174	2.734	61.193
16	car3x	0.385	3.885	186.663
17	GenericPeople	0.409	4.732	199.794
18	tow	0.740	9.843	411.609
19	Palios	1.239	9.282	N/A
20	parallel_while	0.773	14.018	N/A

Tabelle 1: Die Platzierungen der Top 20 des ACM SIGMOD 2014 Programming Contest vom 15. April 2014 [Pcs14]. Durchgestrichene Teams wurden nachträglich wegen falscher Ergebnisse disqualifiziert.

verfügbaren Graph-Datenbanksystemen zu verringern. Insgesamt nahmen 33 internationale Teams am Wettbewerb teil, aus welchen fünf Finalisten und ein Gewinner ermittelt wurden. Keiner der Finalisten baute dabei auf einem bestehenden Graph-Datenbanksystem auf; die Systeme wurden von Grund auf neu implementiert.

In der vorliegenden Arbeit präsentieren wir unser Gewinnersystem des Wettbewerbs (Team AWFY). Tabelle 1 zeigt die Platzierungen der Top 20 vom 15. April 2014. Der Erfolg unseres Systems beruht, neben gutem Engineering, auf den folgenden Entwicklungen: (i) Daten-parallelisierte Graph-Breitensuche, welche Cache-Misses effizient amortisiert, (ii) Heuristiken zur Reduzierung des Suchraums bei Top-k-Anfragen, (iii) schnelles parallelisiertes Laden von textuellen Rohdaten, und (iv) feingranulares Task-Scheduling um Mehrkern-Prozessoren optimal auszulasten. Die in dieser Arbeit beschriebenen Neuentwicklungen lassen sich nach unserer Einschätzung in die meisten bestehenden Graph-Datenbanksysteme integrieren.

In den ersten beiden Teilen dieser Arbeit stellen wir zunächst eine Übersicht unseres Systems für den Wettbewerb und die vier parametrisierten Graph-Anfragetypen vor. Nachfolgend gehen wir auf bekannte Ansätze zur Beantwortung dieser Anfragen ein. Aufbauend auf diesen Ansätzen stellen wir dann unsere Neuentwicklungen vor. Teil 5 beschreibt unsere Daten-parallelisierte Graph-Breitensuche. Teil 6 präsentiert unsere Heuristiken für

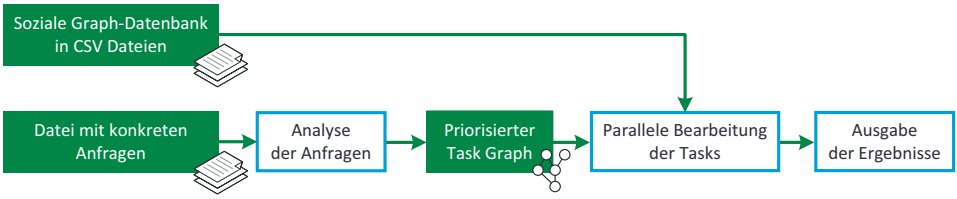


Abbildung 1: Bearbeitung der Wettbewerbsanfragen in unserem System.

Top-k-Anfragen. Im Anschluss zeigen wir, wie in textuellen Rohdaten gespeicherte Graphen schnell geladen werden können und beschreiben den Task-Scheduling-Ansatz unseres Systems. Abschließend fassen wir unsere Erkenntnisse zusammen und geben einen Überblick über zukünftige Aufgaben.

2 Übersicht unseres Systems und des Wettbewerbs

Unser System arbeitet, grob gesehen, in drei Phasen. Am Anfang müssen die konkreten Anfragen sowie der soziale Graph aus den textbasierten Dateien in unser System geladen werden. Die Anfragen analysieren wir und führen sie möglichst schnell aus. Die Verbindung zwischen dem Laden der notwendigen Datenbasis und der Ausführung der Anfragen gelingt uns mit einem priorisierten Task-Graph, der die verschiedenen Schritte parallel ausführt. In einem letzten Schritt werden dann die Ergebnisse in der erwarteten Reihenfolge ausgegeben. Dieser Ablauf ist in Abbildung 1 dargestellt.

Ausschlaggebend für die Bewertung im Wettbewerb war die Gesamtlaufzeit des Systems. Eingesandte Systeme wurden auf einer kleinen Graph-Datenbank mit 1.000 Personen (215 MB), einer mittleren Graph-Datenbank mit 10.000 Personen (7 GB) und einer großen Graph-Datenbank mit 100.000 Personen (23 GB) getestet. Nach Einsendeschluss wurden dann alle Systeme auch auf einer sozialen Graph-Datenbank mit einer Million Personen gebenchmarkt. Die Graph-Datenbanken lagen dabei jeweils in 33 mit dem LDBC *Social Network Data Generator* generierten CSV-Dateien vor [LDB14]. Die zentralen Dateien enthielten personenbezogene Informationen, Interessen der gespeicherten Personen, Freundschaftsbeziehungen und ausgetauschte Nachrichten. Die Freundschaftsbeziehung und der dadurch definierte Graph ist dabei ungerichtet, d.h., wenn Person a mit Person b befreundet ist, so ist auch b mit a befreundet. Dieser Graph dient als Grundlage für alle Anfragetypen. Alle in dieser Arbeit besprochenen Algorithmen operieren daher auf ungerichteten Graphen. Da nicht alle Informationen aus dem LDBC Datensatz zur Beantwortung der Anfragetypen des Wettbewerbs benötigt wurden, konnten nicht benötigte Daten bereits im Ladevorgang verworfen werden.

Um eine Vorberechnung der Ergebnisse zu verhindern, wurden die Anfragen parametrisiert und die Anfrageparameter auf dem Auswertungsserver verdeckt gewählt. Da die Aufgabentypen in ihrer Berechnungskomplexität stark schwanken, wurde die Anzahl der konkreten Anfragen eines Typs mit abnehmender Komplexität hochskaliert. Weil zudem

alle Anfragen von Anfang an zur Verfügung standen, konnte die Ausführung der Anfragen intern umsortiert werden. Bei der Ausgabe der Ergebnisse hingegen musste die ursprüngliche Reihenfolge wieder berücksichtigt werden. Wie in Teil 8 beschrieben, war es uns so möglich eine effiziente Ausführungsreihenfolge der Tasks zu wählen.

Die offizielle Auswertung der Systeme erfolgte auf einem Server des Organisators des Wettbewerbs. Dieser war fest spezifiziert und stellte die Referenzplattform dar. Der Server hatte zwei Intel Xeon E5430 CPUs mit je 4 Kernen ohne Hyper-Threading und 15 GB Hauptspeicher. Die Taktfrequenz der Kerne betrug 2.66 GHz, die letzte Cache-Ebene hatte eine Größe von 12 MB pro CPU. Als Betriebssystem wurde Linux eingesetzt. Die Menge des Hauptspeichers war für die getesteten Graphen mit bis zu einer Million Personen ausreichend, um alle Daten dort zu halten. Keines der Finalsysteme musste während der Ausführung Daten auf die Festplatte auslagern. Die Experimente in diesem Artikel wurden auf eigener Hardware ausgeführt. Unser System hat zwei Intel Xeon X5460 CPUs mit je 4 Kernen (8 Hyper-Threads) und 48 GB Hauptspeicher. Die Taktfrequenz beträgt 3.16 GHz und die letzte Cache-Ebene hat eine Größe von 12 MB pro CPU. Da nur Anfragen für die sehr kleinen 1.000- und 10.000-Personen-Netzwerke veröffentlicht wurden, haben wir darauf basierend entsprechende Anfragen für die 100.000 und 1 Million Netzwerke generiert. Falls nicht anders gekennzeichnet, wurden alle Experimente auf einer Graph-Datenbank mit 100.000 Personen ausgeführt.

3 Anfragetypen

Anfragetyp 1: Kürzeste Pfade entlang von Kommunikationswegen

Im ersten Anfragetyp wird der kürzeste Pfad zwischen zwei Personen a und b in einem ungerichteten Kommunikationsgraphen gesucht. In diesem Graphen sind Kanten zwischen befreundeten Personen enthalten, die miteinander kommuniziert haben. Durch einen zusätzlichen Anfrageparameter x ist definiert, wie oft zwei Personen mindestens miteinander Nachrichten ausgetauscht haben müssen, damit ihre Verbindung in der Pfadsuche berücksichtigt werden soll. Effektiv führt dies dazu, dass jede konkrete Anfrage des ersten Typ auf einem eigenen Subgraphen ausgeführt wird.

Eingabe: query1(a, b, x)

Anfragetyp 2: Suche der Interessengebiete mit den meisten Fans

Im zweiten Anfragetyp werden die k Interessen mit den größten zusammenhängenden Gruppen von Fans gesucht; wir nennen diese Gruppen im Folgenden Interessengruppen. Im sozialen Graph kann jede Person mehrere Interessen haben. Wenn sich mehrere befreundete Personen ein Interessengebiet teilen, bilden sie eine Interessengruppe. Für die Anfrage ist für jedes Interessengebiet nur die größte Gruppe von Personen relevant, in der alle Personen das Interessengebiet teilen und transitiv mit allen anderen Personen in dieser Gruppe befreundet sind. Als weitere Einschränkung müssen die Personen in dieser Gruppe weiterhin nach einem bestimmten Tag b geboren sein. Durch den Parameter b wird somit auch jede konkrete Anfrage des zweiten Typs auf einem eigenen Subgraphen ausgeführt.

Eingabe: query2(k, b)

Anfragetyp 3: Kontaktvorschläge

Das Ziel der Anfragen des dritten Typs ist es die k ähnlichsten Personenpaare in einem Netzwerk zu finden. Als Kriterium für die Ähnlichkeit zwischen zwei Personen dient die Anzahl der geteilten Interessengebiete. Als weitere Einschränkung müssen beide Personen am gleichen Ort p ansässig sein. Im Falle einer Einschränkung auf $p =$ Deutschland sind somit nur Personen relevant, welche innerhalb Deutschlands leben, arbeiten oder studieren. Die geographischen Orte sind dabei als Hierarchien definiert. Neben der örtlichen Einschränkung müssen beide Personen zusätzlich über eine begrenzte Anzahl von Freunden h transitiv miteinander verbunden sein. Diese Distanz wird über den Freundschaftsgraph bestimmt. Die Freunde, über die sie verbunden sind, müssen dabei die Ortseinschränkung nicht zwangsweise erfüllen.

Eingabe: query3(k, h, p)

Anfragetyp 4: Zentrale Personen

Anfragen des vierten Typs suchen die k einflussreichsten Personen in Foren zu einem bestimmten Interessengebiet i. Der Einfluss einer Person bestimmt sich dadurch, wie gut sie mit allen anderen Personen die in diesen Foren teilnehmen befreundet ist. Als Maß wird die *Closeness Centrality* [PKST11] genutzt:

$$\frac{(r(p)-1)*(r(p)-1)}{(n-1)*s(p)}$$

Der Wert von $r(p)$ gibt dabei für eine Person p an, wie viele Personen von ihr aus im Freundschaftsgraph erreicht werden können. $s(p)$ gibt für eine Person die Summe über die Distanzen zu allen anderen erreichbaren Personen an. n ist die Anzahl der Personen im Graph. Falls der Zähler oder Nenner 0 sind, ist der Wert der *Closeness Centrality* auch 0. Es wird wiederum nur ein Subgraph des Freundschaftsgraph betrachtet, welcher definiert ist durch Freunde, die auch Mitglieder in einem der Foren für ein Interessengebiet i sind. Eine statische Vorberechnung der Distanzen ist daher nicht möglich.

Eingabe: query4(k, i)

4 Bekannte Ansätze

Die vier verschiedenen Anfragetypen lassen sich großteils auf wenige generische Graphprobleme abbilden: Anfragetyp 1 ermittelt kürzeste Pfade, Anfragetypen 2 berechnet Zusammenhangskomponenten, Anfragetypen 3 untersucht Personenpaare in *k-hop*-Nachbarschaften und berechnet die Schnittmenge von deren Interessen und Anfragetypen 4 entspricht der Berechnung des kürzesten Pfades zwischen allen Paaren von Personen (*All Pairs Shortest Path*).

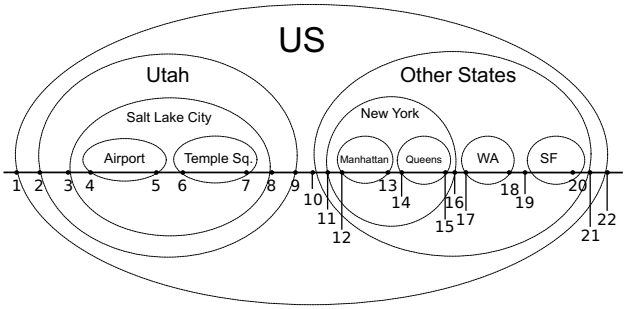


Abbildung 2: Beispiel eines *Nested Interval Encoding* für Orte

Eine sehr effizienter Algorithmus um kürzeste Pfade zwischen zwei Knoten in einem ungerichteten Graphen zu berechnen ist bidirektionale Breitensuche [ES11]. Hierbei wird jeweils am Start- und Ziel-Knoten eine Breitensuche gestartet. Die beiden Breitensuchen werden abwechselnd so lange ausgeführt, bis sie sich in mindestens einem Knoten treffen. Die Summe der Tiefen beider Breitensuchen ergibt dann die Länge des kürzesten Pfades zwischen den beiden Knoten. Breitensuchen sind weiterhin die effizienteste Lösung um Zusammenhangskomponenten in einem ungerichteten Graphen zu finden. *k-hop*-Nachbarschaften, also alle Knoten mit Distanz $\leq k$ vom Startknoten, entsprechen genauso einer beschränkten Breitensuche, die nach spätestens k Schritten terminiert. All Pairs Shortest Path entspricht einer Breitensuche von jedem Knoten aus. Zusammengefasst lassen sich also alle Anfragetypen mit einer Variante der Breitensuche lösen.

Als potenziell teure Probleme verbleiben noch das Ermitteln der Elemente im Schnitt von zwei Interessen-Mengen und die Überprüfung sowie das Feststellen der Ortszugehörigkeit von Personen im dritten Aufgabentyp. Wir verwenden dafür bekannte Ansätzen aus der Literatur. Zur Bildung von Schnittmengen nutzten wir einen neuen Daten-parallelen Ansatz von Lemire et al. [LBK14], welcher SIMD-Instruktionen auf Intel Prozessoren nutzt, um die Schnittmenge zwischen zwei sortierten Listen zu bilden. Das Sortieren der Interessenslisten pro Person konnte günstig während des Ladens der Daten durchgeführt werden. Um die Ortszugehörigkeit möglichst effizient zu testen, nutzten wir das *Nested Interval Encoding* [Tro05] um die Hierarchie der Orte zu speichern. Diese Technik weist jedem Ort einen Bereich auf einem Zahlenstrahl so zu, dass alle untergeordneten Orte innerhalb dieses Bereichs liegen. Jeder Ort ist somit durch einem Anfangs- und End-Zahlenwert beschrieben. Die Überprüfung, ob ein Ort Teil eines anderen Orts ist beschränkt sich somit auf zwei Vergleiche von Intervallgrenzen. Ein Beispiel hierfür haben wir in Abbildung 2 dargestellt. Falls beispielsweise in einer Anfrage die Zugehörigkeit des Orts *New York* zu *Utah* überprüft wird, so genügt es die Intervallgrenzen von New York (hier durch das Intervall $[11, 16]$ codiert) mit denen von Utah (mit dem Intervall $[2, 9]$) zu vergleichen. Da bereits $11 > 9$ ist, liegt New York nicht in Utah.

Im Folgenden gehen wir genauer auf die bekannten Ansätze zur Durchführung einer Breitensuche ein.

Den klassischen Algorithmus für eine Breitensuche[CSRL01] zeigen wir in Abbildung 3.

```
1  Input: G, s
2  seen ← {s}
3  visit ← {s}
4  visitNext ← ∅
5
6  while visit ≠ ∅
7      for each v ∈ visit do
8        for each n ∈ neighbors_v do
9            if n ∉ seen then
10             seen ← seen ∪ {n}
11             visitNext ← visitNext ∪ {n}
12             Mögl. Knotenspez. Berechn.
13     visit ← visitNext
14     visitNext ← ∅
```

Abbildung 3: Klassischer BFS Algorithmus

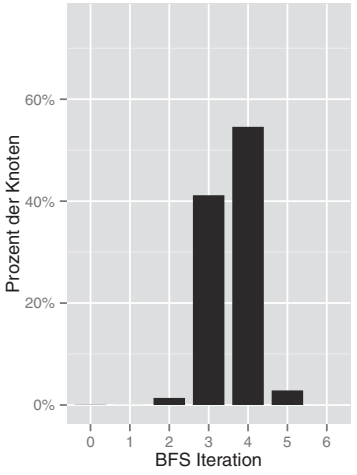


Abbildung 4: Anzahl der entdeckten Knoten pro Iteration

Als Eingabe erwartet die Breitensuche eine Graph-Struktur $G = (V, E)$ bestehend aus einer Knotenmenge V und einer Kantenmenge E und den Startknoten der Suche s. Während der Ausführung wird ausgehend von bekannten Knoten versucht neue bisher nicht besuchte Knoten zu erreichen. Um zu markieren, dass ein Knoten bereits bekannt ist, wird er bei erstmaligem Entdecken in der $seen$-Liste gespeichert. Initial enthält diese nur den Startknoten. Außerdem wird jeder Knoten in eine FIFO-Warteschlange eingereiht. Die Ausführung ist in Runden aufgeteilt. In jeder Runde wird die Liste der aktuell zu besuchenden Knoten $visit$ abgearbeitet und es wird versucht über die ausgehenden Kanten dieser Knoten neue Knoten zu entdecken. Alle neu entdeckten Knoten werden zur $visitNext$-Liste hinzugefügt. Diese Liste wird dann in der nächsten Iteration abgearbeitet. Die Distanz zum Startknoten ergibt sich aus der Anzahl der Iterationen, die benötigt werden, um einen Knoten zu erreichen.

Basierend auf diesem Algorithmus wurden verschiedene Ansätze entwickelt, um die Breitensuche zu beschleunigen. Diese können zur Vereinfachung in zwei Gruppen aufgeteilt werden: (i) Ansätze, die sich auf maximale Performanz auf einem Prozessorkern fokussieren und (ii) Ansätze die eine einzelne Breitensuche mit Hilfe mehrerer Prozessorkerne beschleunigen. Da die Parallelisierung einer einzelnen Breitensuchen immer zusätzliche Kosten für Synchronisation und Kommunikation bedeutet [BM11], hilft dies nur falls die Kerne sonst nicht ausgelastet werden können. Da wir in unserem Anwendungsfall um mehrere Größenordnungen mehr Breitensuchen als Prozessorkerne durchführen müssen, ist es sinnvoller mehrere sequenzielle Breitensuchen auf die Kerne zu verteilen. Allein Anfragetyp 4 muss potenziell von jedem Knoten aus eine Breitensuche starten.

Der schnellste sequenzielle Ansatz eine einzelne Breitensuche zu berechnen wurde von Beamer et al. [BAP12] vorgestellt. Die Verbesserung beruht auf der Beobachtung, dass in den letzten Iterationen einer BFS die Warteschlange $visit$ größer ist als die Menge

der noch nicht besuchten Knoten. Nur sehr wenige Nachbarn der Knoten in der Warteschlange sind noch nicht besucht, aber trotzdem müssen alle Nachbarn überprüft werden. Dies macht das Finden der letzten Knoten sehr teuer. Diese Beobachtung trifft auch auf den sozialen Graphen im Wettbewerb zu. In Abbildung 4 ist dargestellt, in welchen Iterationen jeweils wie viel Prozent der Knoten des 100.000-Personen-Netzwerks gefunden wurden. Nach der vierten Iteration sind nur noch 10% der Knoten nicht gefunden, aber die $visit$-Liste für die fünfte Iteration enthält 55% der Knoten im Graph. Um die Suchkosten für die letzten Knoten zu senken, schlagen die Autoren eine *bottom-up*-Variante vor, die das Vorgehen der Suche umkehrt. Anstatt ausgehend von den Knoten in der $visit$-Liste nach neuen Knoten zu suchen, überprüfen sie ob Knoten, die noch nicht gefunden wurden, einen Nachbarn haben, der in der $visit$-Liste enthalten ist. Dadurch wird in den letzten Iterationen die Performanz von der Anzahl der fehlenden Knoten bestimmt. Da dies nur Sinn macht, wenn die Anzahl der gefundenen Knoten abnimmt, wird zusätzlich eine Heuristik eingeführt um den Zeitpunkt des Wechsels auf die *bottom-up*-Variante zu entscheiden. Insgesamt schaffen die Autoren mit ihrem Ansatz auf sozialen Netzwerken eine Laufzeitverbesserung von ungefähr Faktor 4 [BAP12].

Ein großes verbleibendes Problem sind aber immer noch die vielen unregelmäßigen Speicherzugriffe, die ein effizientes *Prefetching* der Daten durch den Prozessor unterbinden. Da die Warteschlange nicht nach Knotenreihenfolge, sondern in Reihenfolge des Einfügens sortiert ist, entstehen beim Zugriff auf die Nachbarliste Sprünge im Speicher. Auch bei der Kontrolle jedes Nachbarn müssen die entsprechenden $seen$-Einträge aus nicht vorhersehbaren Speicherbereichen geladen werden. Diese Zugriffe schränken die Performanz signifikant ein, was besonders bei großen Graphen die nicht mehr in den Cache passen zu Geschwindigkeitseinbrüchen führt, da die Ausführungszeit in diesem Fall durch die Speicherzugriffslatenz bestimmt wird.

5 Daten-Parallelisierte Graph-Breitensuche

Um den Wettbewerb zu gewinnen, haben wir nach neuen Ansätzen zur Beschleunigung der Breitensuche gesucht. Eine zentrale Erkenntnis ist, dass Knoten in einem sozialen Graphen sehr dicht miteinander verknüpft sind. Die Breitensuche in Abbildung 4 beispielsweise terminiert nach nur sechs Iterationen. Eine weitere Beobachtung ist, dass die meisten Knoten in nur zwei Iterationen entdeckt werden. Daraus lässt sich ableiten, dass sich die Warteschlangen mehrerer Breitensuchen jeweils stark überlappen, da die meisten Breitensuchen über 80% der Knoten in diesen Iterationen entdecken. Sobald verschiedene Breitensuchen den gleichen Knoten in der gleichen Iteration entdecken, ist die Wahrscheinlichkeit, dass sie auch weitere Knoten zusammen entdecken, sehr groß. Es ergibt sich somit ein Potential diese Breitensuchen zu bündeln.

Basierend auf dieser Erkenntnis haben wir einen Ansatz entwickelt der erkennt, wann Breitensuchen an Knoten überlappen, diese dann zusammen ausführt und somit einen signifikanten Geschwindigkeitsgewinn erzielt. Die Hauptprinzipien beim Entwurf des neuen Ansatzes waren: (i) die hohen Kosten für die Zugriffe auf die Datenstrukturen zu reduzieren, indem man die gebündelten Suchen zusammen abarbeitet und somit die Speicher-

```
1  Input: G, 𝔹, S
2  seen[s_{b_i}] ← {b_i} for all b_i ∈ 𝔹
3  visit[s_{b_i}] ← {b_i} for all b_i ∈ 𝔹
4  visitNext[v] ← ∅ for all v ∈ G
5  queue_empty ← false
6
7  while queue_empty ≠ false
8      queue_empty ← true
9      for all v ∈ G
10       if visit[v] ≠ ∅
11           for each n ∈ neighbors_v
12             𝔻 ← visit[v] \ seen[n]
13             if 𝔻 ≠ ∅
14                 queue_empty ← false
15                 visitNext[n] ← visitNext[n] ∪ 𝔻
16                 seen[n] ← seen[n] ∪ 𝔻
17                 Mögl. Knotenspez. Berechn.
18     visit ← visitNext
19     visitNext[v] ← ∅ for all v ∈ G
```

Abbildung 5: Bitparallele Breitensuche

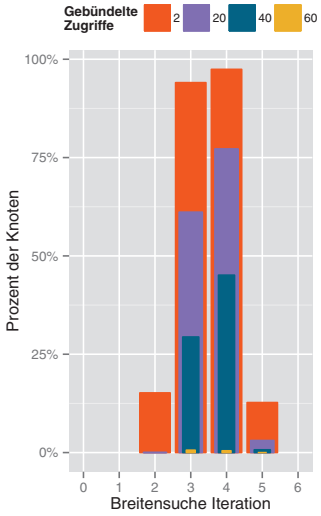


Abbildung 6: Anteil gebündelter Knotenzugriffe pro Iteration. Gruppiert nach Anzahl der gebündelten Breitensuchen.

zugriffslatenz der Cache-Misses amortisiert und (ii) die Bündelung auszunutzen, um die Ausführung durch den Einsatz von Bit-Operationen weiter zu beschleunigen. Der von uns entworfene Algorithmus ist in Abbildung 5 gezeigt.

Die Grundstruktur des Algorithmus im Vergleich zu einer klassischen Breitensuche bleibt erhalten. In jeder Iteration wird weiterhin versucht ausgehend von den entdeckten Knoten neue Knoten zu finden. Die Datenstrukturen und Operationen darauf müssen jedoch angepasst werden, um die Bündelung der Breitensuchen effizient zu ermöglichen. Statt eines einzelnen Startknotens als Eingabeparameter gibt es nun eine Menge von Breitensuchen $\mathbb{B}$ und entsprechend eine Menge von Startknoten S. Jede Breitensuche ist dabei durch einen Index b_i codiert, über den auch auf den entsprechenden Startknoten s_{b_i} zugegriffen werden kann. Für die Breitensuchen müssen dabei jeweils die Informationen, welche Knoten bereits gesehen wurden und welche Knoten in der Iteration bearbeitet werden, gespeichert werden. Die bisherigen binären $seen$- und $visit$-Listen reichen dafür nicht aus.

Um diese Informationen für mehrere Breitensuchen zu speichern, gibt es intuitiv zwei Möglichkeiten. Es kann zum Einen für jede Breitensuche separat jeweils eine binäre $seen$- und $visit$-Liste genutzt werden. Zum Anderen können weiterhin zwei $seen$- und $visit$-Listen genutzt werden, die zu jedem Eintrag die Menge der betreffenden Breitensuchen speichern; falls Knoten a z.B. schon von den BFS b_1, b_2 und b_3 gesehen wurde, dann sähe der entsprechende $seen[a]$ Eintrag so aus: $\{b_1, b_2, b_3\}$. Während die Implementierung der ersten Variante sehr leicht ist, hilft sie nicht das Problem der vielen Speicherzugriffe durch Bündelung signifikant zu reduzieren. Die Zugriffe auf die Nachbarschaftsinformationen

für gemeinsame Breitensuchen können zwar gebündelt werden, aber beim Zugriff auf die $seen$- und $visit$-Informationen der Nachbarn wären weiterhin für jede Breitensuche separate Speicherzugriffe notwendig. Die zweite Variante bietet unter diesem Aspekt mehr Potenzial um Kosten zu sparen.

Ein speichereffizienter und günstiger Weg die Mengen in den jeweiligen Einträgen zu speichern sind *Bitsets*. Für jede Breitensuche wird dabei die Information, ob sie den jeweiligen Knoten gesehen hat, oder besuchen wird mit einem Bit gespeichert. Jede Breitensuche besitzt eine einzigartige feste Bitposition. Um beispielsweise zu markieren, dass BFS b_3 den Knoten a besucht hat, wird im Bitset des Knoten das dritte Bit auf 1 gesetzt. Dies passiert z.B. bei der Initialisierung des Algorithmus in Zeile 2 des Algorithmus. Die Vereinigungsoperation $\cup$ kann dabei durch den binären oder-Operator $|$ mit einem entsprechenden Bitvektor z.B. 00001000 für b_3 in einem Bitset mit acht Einträgen implementiert werden.

Bei der Umsetzung der Warteschlangen muss berücksichtigt werden, dass die Breitensuchen gebündelt werden sobald sie die gleichen Knoten in der gleichen Iteration erreichen. Falls beispielsweise BFS b_1 den Knoten x von Knoten n aus erreicht und BFS b_2 den Knoten x von m aus, würden ohne Anpassungen zwei Einträge in die Warteschlange eingefügt werden. Um die BFS zu bündeln ist es aber erforderlich einen einzigen Warteschlangeneintrag $\{b_1, b_2\}$ für den Knoten x in der Warteschlange zu haben. Die effizienteste Art dies zu implementieren ist es statt einer klassischen Warteschlange, in die Einträge eingefügt, werden, eine Liste zu nutzen, die für jeden Knoten einen festen Eintrag hat. Um einen Knoten für eine BFS nun für die nächste Runde einzureihen, wird beim entsprechenden Eintrag des Knoten in der Liste das Bit für diese Breitensuche gesetzt. Somit existiert nach einer Iteration immer genau ein Eintrag für jeden Knoten, der angibt welche Breitensuchen diesen Knoten in der aktuellen Iteration entdeckt haben und somit für welche in der nächsten Iteration von diesem Knoten aus weiter gesucht werden muss. Im Unterschied zu der ursprünglichen Warteschlange ist kein FIFO-Verhalten mehr garantiert, dies wird aber für die Korrektheit des Algorithmus nicht benötigt. Es muss nur gesichert sein, dass die Warteschlangen zwischen Iterationen getrennt sind. Der algorithmische Nachteil dieser Variante ist, dass in einer Iteration immer alle Einträge der Knoten des Graphen in der Liste überprüft werden müssen, um die zu finden, die bearbeitet werden müssen. In der Praxis ist diese sequenzielle Suche aber sehr schnell; sie ist neben den Kosten der anderen Operation des Algorithmus vernachlässigbar.

Die angepassten Datenstrukturen erfordern auch, dass die Operationen auf ihnen angepasst werden. Die Ausführung des Algorithmus geschieht nicht mehr als individuelle Binäroperationen, sondern diese werden in die entsprechenden Mengenoperationen umgesetzt. Die Überprüfung zum Beispiel, ob ein Knoten bereits gesehen wurde, war bisher einfach eine Überprüfung ob der entsprechende Eintrag in der $seen$ Liste den Wert $true$ hat. Mit unserem neuen Ansatz muss dies nun äquivalent für eine Menge an Breitensuchen gemacht werden. Wichtig dabei ist nur die Breitensuchen zu berücksichtigen, die an dem jeweiligen Knoten auch aktiv sind. Um nun herauszufinden, welche Breitensuchen den Knoten neu entdecken, muss die Differenz zwischen der Menge der aktiven Knoten und der Menge des $seen$ Eintrags des Knoten bestimmt werden. Die verbleibende Menge an Breitensuchen hat dann diesen Knoten zum ersten Mal gesehen und es müssen entsprechend die $seen$ Einträge aktualisiert (Zeile 16) und die $visitNext$ Liste für die nächste

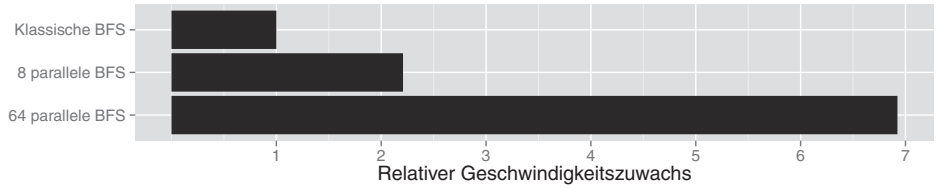


Abbildung 7: Laufzeitverbesserung mit 8 beziehungsweise 64 gebündelten Breitensuchen

Iteration im diese BFS erweitert werden (Zeile 15).

Diese Mengenoperationen wie Differenz, Vereinigung, etc. können, da die Mengen durch Bitsets repräsentiert werden, effizient mit or-, and- und $andnot$-Instruktionen umgesetzt werden. Um die Instruktionen nutzen zu können, dürfen die Bitsets aber nicht größer als die Register der CPU sein. Das Wettbewerbssystem unterstützte zwar 128 bit breite SSE Register, aber unterstützt auf diesen nur einen zu sehr eingeschränkten Befehlssatz. Aus diesem Grund ist die Bitsetgröße auf 64 bit und somit 64 parallele Breitensuchen beschränkt. Durch die Bündelung von maximal vielen Breitensuchen entstehen zwei Vorteile: (i) die Anzahl der Instruktionen für die bitparallele BFS ist unabhängig von der Registergröße und somit werden die Kosten bei breiteren Registern besser verteilt und (ii) mit der Anzahl der Breitensuchen steigt auch die Wahrscheinlichkeit, dass mehrere an einem Knoten überlappen. Gerade in den Iterationen in denen viele Knoten entdeckt werden, können so noch mehr Breitensuchen gebündelt werden.

Um die Effektivität der Bündelung zu prüfen, haben wir bei der 64-fach Daten-Parallelen Breitensuchen gemessen auf wie viele Knoten pro Iteration gebündelt zugegriffen wird. Die Messungen sind dabei unterteilt nach Knoten, die von mindestens 2, 20, 40 oder 60 gebündelten Breitensuchen besucht wurden. Wie in Abbildung 6 zu sehen ist, können gerade in den Iterationen, in denen auf viele Knoten zugegriffen wird, die Breitensuchen oft stark gebündelt werden. Abbildung 7 zeigt dementsprechend auch, dass wir mit der Bündelung gegenüber der klassischen Variante die Laufzeit um über Faktor 6 reduzieren konnten. Als weiteren Vergleich haben wir die Laufzeit zu Berechnung der Distanzsumme für jeden Knoten mit dieser Breitensuchen mit der Laufzeit des in Neo4j integrierten Algorithmus verglichen. Unsere Laufzeit war um über Faktor 100 schneller. Nach dem Wettbewerb haben wir diesen Ansatz stark weiter verfeinert und auch auf breiteren Registern getestet. Diese Ergebnisse wurden in einem separaten Artikel veröffentlicht [TKC+14].

6 Heuristiken für Top-k-Anfragen

Neben der Beschleunigung der Breitensuche gibt es für die *Top-k*-Anfragen noch die Möglichkeit den Suchraum mit Hilfe von oberen und unteren Schranken, die mit dem bisherigen besten Ergebnis verglichen werden können, zu verkleinern. Dies hilft Breitensuchen zu sparen oder früher abzubrechen.

Der Benchmark enthält mit Anfragetypen 2–4 einige solcher Anfragen. Für diese haben

Interessen	Metadaten		Ergebnis
	jüng. Geb.	#Pers.	
Vorl. Top-1			
Fußball	02.06.1998	2.600	2.460
Kandidaten			
Bratwurst	06.12.1999	2.500	~~2.430~~
Urlaub	08.07.1996	~~1.400~~	—
Auto	~~13.11.1992~~	~~1.350~~	—

Tabelle 2: Beispiel der Heuristik für Anfragetyp 2 mit den Parametern $k = 1$ und $b = 01.01.1993$. Interessen mit durchgestrichenen Werten werden eliminiert.

wir verschiedene Heuristiken entwickelt, die günstig obere und untere Schranken liefern. Im Folgenden stellen wir die Heuristiken für jeden Anfragetyp vor.

In Anfragen vom Typ 2 werden möglichst große Interessengruppen mit Personen unterhalb einer Altersgrenze b gesucht. Für jedes Interesse bildet die Gesamtzahl der Personen, die sich dafür begeistern, eine obere Schranke für die Größe von dessen maximaler Interessengruppe. Gleichzeitig sind die Interessen unterschiedlich über die verschiedenen Altersgruppen verteilt. Falls das Alter der jüngsten Person, die ein Interesse hat, höher ist als von einer Anfrage gefordert, kann für dieses sofort die Maximalgröße 0 bestimmt werden. Beide Metriken, die Anzahl der Personen und die jüngste Person pro Interesse, lassen sich günstig mit einem Scan aller Personen ermitteln. Tabelle 2 zeigt anhand eines Beispiels das Potenzial dieses Verfahrens. Es zeigt die aktuellen Zwischenausführung während der Ausführung einer *Top-1* Anfrage die sich auf Personen die nach dem $01.01.1993$ geboren wurden beschränkt. Beim Interesse *Bratwurst* helfen die Schranken noch nicht die exakte Berechnung zu vermeiden. Die 2.500 Personen, die dieses Interesse teilen reichen theoretisch aus um die bisher größte Gruppe von 2.460 zu übertreffen. Auch mit Hilfe der Geburtstagsinformation kann keine ausreichende Schranke ermittelt werden. Erst nach Berechnung des exakten Wertes von 2.430 wird ersichtlich, dass es dieses Interesse nicht Teil der Top-1 ist. Für die anderen Interessen hingegen genügt jeweils eine Überprüfung der vorberechneten Metadaten, um eine exakte Berechnung zu vermeiden. Somit kann jeweils eine Breitensuche eingespart werden. Um möglichst früh eine starke Schranke zu erhalten, werten wir die Interessen in der Reihenfolge gemäß der Anzahl der Personen aus Abbildung 8 zeigt die Wirksamkeit dieser Heuristiken anhand von Messungen auf dem 100.000-Personen-Netzwerk. Gemessen wurde jeweils die Gesamtlaufzeit inklusive dem Laden der benötigten Dateien. Die Ausführung erfolgte nur auf einem CPU-Kern. Allein mit der Heuristik basierend auf der Anzahl der Personen kann die Laufzeit auf unter fünf Sekunden reduziert werden. Kombiniert mit Schranken basierend auf der jüngsten Person für jedes Interesse sinkt die Laufzeit auf 2.059 ms. Davon werden 9 ms zur Generierung der Metadaten für die Heuristiken benötigt. Die Ermittlung der exakten Werte für die Interessen, die nicht übersprungen werden konnten, dauerte 1.778 ms. Der Rest der Laufzeit wurde zum Laden der Eingabedaten benötigt.

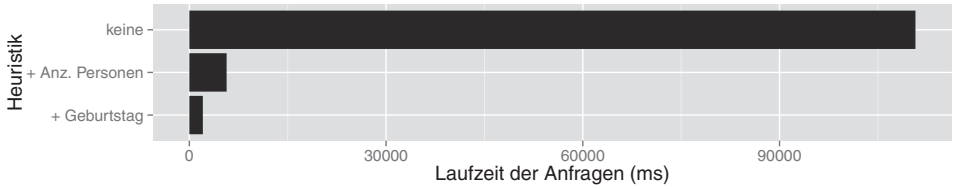


Abbildung 8: Gesamtlaufzeit der Auswertung von zehn Anfragen vom zweiten Typ bei gestaffelter Aktivierung der Heuristiken. Jede Variante berechnet jeweils nur die minimal benötigten Metadaten.

Anfrage 3 beschäftigt sich mit Personenpaaren, die sich an einem vorgegebenen Ort aufhalten, sich über eine begrenzte Anzahl von Freunden kennen und die meisten Interessen miteinander teilen. Bei dieser Anfrage gibt es zwei teure Schritte. Zum einen müssen alle Freundschaftspaare bestimmt werden, die diese Kriterien erfüllen und zum Anderen muss für jedes Personenpaar überprüft werden, wie viele gemeinsame Interessen sie haben. Für diesen zweiten Schritt kann eine simple, aber effektive Heuristik genutzt werden: Für jedes Personenpaar bildet die Anzahl der Interessen einer der Personen eine obere Schranke für die Anzahl der gemeinsamen Interessen. Diese Schranke kann während der Auswertung mit den bisherigen besten Ergebnissen zur Anzahl der gemeinsamen Interessen verglichen werden. Ist in diesem Vergleich die Schranke kleiner, ist es nicht nötig die genaue Größe der Schnittmenge zu bestimmen, da dieses Paar nicht Teil der Top-k-Ergebnisse sein kann. Um früh ein möglichst guten Top-k-Grenzwert zu erhalten und so möglichst viele andere Paare überspringen zu können werten wir die Personenpaare absteigend geordnet nach der Anzahl ihrer Interessen aus.

Das meiste Potenzial zur Reduzierung der Gesamtlaufzeit steckt jedoch im teuren Anfragetyp 4. Zur Berechnung der Zentralität muss von jeder Person die Distanz zu allen anderen erreichbaren Personen berechnet werden. Mit einer Heuristik, die eine verlässliche günstige untere Schranke auf die Summe der Distanzen zu allen anderen Personen liefert, ließen sich potenziell durch den Vergleich mit den aktuellen Top-k-Werten viele Distanzberechnungen vermeiden. In der Literatur existieren probabilistische Ansätze die Summe günstig mittels Hash-Sketches abzuschätzen [PKST11]. Diese Verfahren können aber keine Worst-Case-Garantien für die Fehler der Ergebnisse liefern. Somit können sie auch nicht zur Berechnung von Schranken genutzt werden, da sonst eventuell Personen fälschlicherweise übersprungen werden und das Ergebnis verfälscht wird. Wir adaptieren den Ansatz und entwickeln eine neue Abschätzung, die im Schnitt zwar schlechter ist, dafür aber verlässlichere untere Schranken für die Distanzsumme liefert, welche zur Berechnung der Closeness Centrality benötigt wird.

Als Basis unseres Verfahrens analysieren wir zunächst die Struktur des probabilistischen Distanzschätzungsalgorithmus [PKST11]. Die Grundintuition ist, dass in einem ungerichteten Graph die Mengen der Knoten, die von einem Knoten aus mit x Schritten erreicht werden können, der Menge der Knoten entspricht, die alle seine Nachbarn mit $x-1$ Schritten erreichen. Vom Nachbar zum Knoten ist es schließlich genau ein Schritt weiter. Dies nutzt der Algorithmus aus, indem pro Knoten jeweils die Vereinigung der von allen Nach-

barn in der letzten Runde erreichten Knoten gebildet wird. Der Unterschied in der Größe zwischen den Mengen eines Knotens in der Iteration $x - 1$ von der in Iteration x gibt die Anzahl der Personen an, die mit genau x Schritten erreicht werden können. Um die Distanzsumme pro Knoten zu ermitteln, wird die Menge der erreichbaren Knoten vor der ersten Iteration jeweils nur mit sich selbst gefüllt. In Iteration 1 wird dann für jeden Knoten die Vereinigung der initialen Mengen aller seiner Nachbarn gebildet und die Größe dieser Menge dann mit der der intitalen Menge verglichen. Dies ergibt die Knoten, die genau einen Schritt entfernt sind. In Iteration 2 dann für jeden Knoten die Vereinigung aller von den Nachbarn in Iteration 1 erreichten Knoten, etc. Indem man diese Distanzen aufsummiert bis in einer Iteration keiner der Knoten mehr eine neue Person erreicht hat, kann jeweils die Distanzsumme für jeden Knoten ermittelt werden. Im Kontrast zu einer einfachen Breitensuche wäre es aber prohibitiv teuer die Vereinigungen der Nachbarmengen exakt zu berechnen, da in den letzten Iterationen für jeden Knoten die Zahl der Elemente seiner Nachbarknotenmenge in der Größenordnung der Zahl von Knoten im Graphen ist. Um die damit verbundenen hohen Speicher- und Rechenkosten zu vermeiden, werden approximative Datenstrukturen, die den Speicherbedarf reduzieren genutzt. Die Anforderungen dabei sind, dass sie die Vereinigungs-Operation günstig unterstützen müssen und die Größe der Menge abgeschätzt werden kann. Dieses zweite Kriterium wird z.B. von Bloom-Filtern [Blo70] nicht unterstützt. Stattdessen können Strukturen wie *FM-Sketches* [FNM85] oder *HyperLogLog* [HNH13] genutzt werden. Da diese aber nur probabilistische Werte für die Größe der Mengen bestimmen können, sind sie nicht zur Berechnung exakter Ergebnisse geeignet.

Die Basis unseres Ansatzes ist es die Schätzung der Menge der über die Nachbarn erreichten Knoten noch weiter zu vereinfachen, indem wir o.B.d.A. annehmen, dass diese Mengen unabhängig sind. Wenn dieses unabhängig sind, können einfach ihre einzelnen Größen zusammenaddiert werden, um die Größe der Vereinigung der Mengen zu berechnen. Somit werden als einzige Information immer nur die Größe der Menge benötigt. Auf das Verfahren angewandt bedeutet dies, dass statt der probabilistischen Mengen dann nur auch jeweils die Größe gespeichert wird. Die Distanzschätzung, die mit dieser Anpassung berechnet wird, ist eine strikte Obergrenze, auch wenn die Unabhängigkeitsannahme verletzt ist. Wenn sich die Mengen der Nachbarn überlappen würden, wäre die Anzahl der erreichten Knoten pro Iteration geringer, damit wäre die Anzahl der Knoten, die mit geringer Distanz erreicht werden auch kleiner und somit die Distanzsumme größer als unsere Schätzung. In einem Iterationsschritt werden die Summe der Schätzungen der Nachbarn mit der Schätzung für den eigenen Knoten der letzten Iteration verglichen und die Differenz ist die Anzahl der neu erreichten Knoten. Die Abschätzung der Distanzsumme kann auch zur Wahl der Auswertungsreihenfolge der Knoten genutzt werden. Die Hoffnung dabei ist, dass so durch gute vorläufige Top-k Werte möglichst früh viele Breitensuchen mit Hilfe der Schätzung übersprungen werden können.

Als letzte Optimierung um diese Heuristik zu verbessern, haben wir die Information der Breitensuchen der Knoten, die nicht übersprungen werden können, genutzt, um die Schätzungen weiter zu verbessern. Dazu speichern wir bei der Berechnung der Distanzschätzung für jeden Knoten, wie viele Knoten jeweils mit x Iteration geschätzt erreicht werden können. Falls für einen Knoten nun eine Breitensuche komplett ausgeführt wird, können

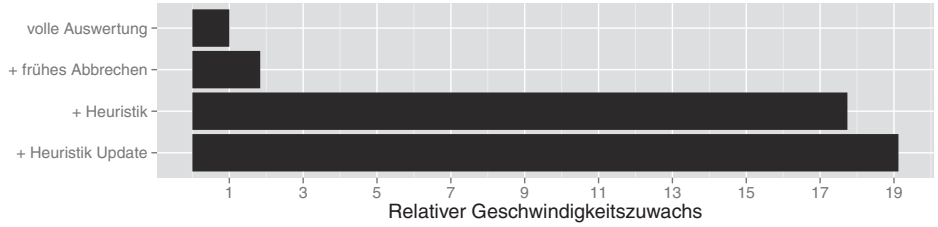


Abbildung 9: Relative Geschwindigkeitszuwächse durch Heuristiken bei Anfragetyp 4

diese Schätzungen dann durch die echten Werte ersetzt werden. Bei der Neuberechnung der Schätzung für einen seiner Nachbarn können nun diese exakten Werte genutzt werden, um die Schätzung zu verbessern.

Zusammengefasst berechnen wir für jeden Knoten zuerst eine Schätzung, wie viele Knoten jeweils mit x Schritten erreicht werden können, speichern diese und berechnen eine erste Schätzung für die Distanzsumme. Danach beginnt die Auswertung der Knoten. Bevor ein Knoten ausgewertet wird, wird überprüft ob er mit Hilfe der ursprünglichen Schätzung übersprungen werden kann. Ist dies nicht der Fall wird die Schätzung neu berechnet mit der Hoffnung, dass für einige seiner Nachbarn die genauen Werte der mit x Schritten erreichbaren Knoten statt der Schätzungen zur Verfügung stehen und somit eine bessere Schätzung erreicht werden kann. Falls mit dieser neuen Schätzung der Knoten auch nicht übersprungen werden kann, muss die Breitensuche gestartet werden. Die Laufzeitverbesserungen dieser aufeinander aufbauenden Verbesserungen haben wir gemessen und in Abbildung 9 dargestellt. Für die Wettbewerbsanfragen vom Typ 4 konnten wir auf dem 100.000-Personen-Datensatz somit eine Verbesserung der Laufzeit von Faktor 19 erreichen.

7 Beschleunigtes Laden von Rohdaten

Auf Grundlage der in [MRS+13] beschriebenen Techniken um das Laden von textbasierten Rohdaten in relationalen Hauptspeicher-Datenbanksystemen zu beschleunigen haben wir neue Methoden entwickelt um Graphdaten effizient in unser Analysesystem zu laden. Die LDBC-Daten des sozialen Graphen lagen beim Wettbewerb als CSV Dateien vor. Dies ist auch in realen Anwendungsszenarien häufig der Fall, denn CSV ist weiterhin ein beliebtes Speicherformat. Die Vorteile von CSV Dateien liegen in ihrer Lesbarkeit, da alle Daten textuell gespeichert sind und daher in einem Texteditor gelesen und verändert werden können. Des Weiteren sind CSV Dateien, anders als proprietäre Binärformate, sehr portabel. Allerdings können textuell gespeicherte Daten nicht effizient analysiert werden. Die Zeilen einer CSV Datei müssen zunächst entlang des Feldtrennzeichens, häufig ",", aufgespalten werden (*Parsing*). Des Weiteren müssen für manche Datentypen die textuellen Daten noch in ein maschinenverständliches Format, z.B. Integer, umgewandelt werden (*Deserialization*). Beides, Parsing und Deserialization sind auf modernen Prozessoren

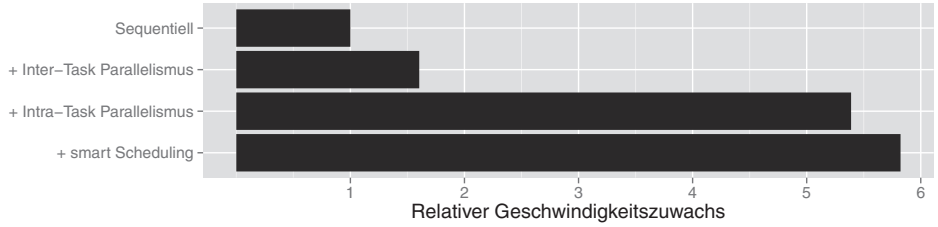


Abbildung 10: Geschwindigkeitszuwachs bei der Auswertung der Benchmarkanfragen beim Hinzufügen der verschiedenen Techniken

sehr teuer. Naive Implementierungen vergleichen jedes Byte mit dem Feld- und Zeilen-Trennzeichen, was zu einem unregelmäßigen Kontrollfluss und daher zu vielen *Branch-Misses*[2] führt. Daten-paralleles Parsing wie in [MRS+13] beschrieben verarbeitet mehrere Vergleiche in einer Instruktion. Mit der Intel SSE 2 Instruktion _mm_cmpeq_epi8 können etwa 16 Byte mit einem Vergleichs-Byte in einer Instruktion verglichen werden. Die resultierende Bitmaske liefert dann die Stellen zurück, an denen das Vergleichs-Byte gefunden wurde. Dies erzeugt deutlich weniger Branch-Misses und führt daher zu einer höheren Performanz. Für das Wettbewerbssystem konnten wir so die Ladegeschwindigkeit um über 60% gegenüber naivem Parsing und Deserialization mit Methoden aus der Standardbibliothek (z.B. atol) steigern.

Ebenfalls, wie in [MRS+13] beschrieben, wird das Laden einer einzelnen Datei in mehrere Tasks aufgespalten, welche dann von mehreren Prozessorkernen parallel bearbeitet werden. Je nach Geschwindigkeit der Datenquelle kann so die Geschwindigkeit des Ladens mit der Anzahl der verwendeten Prozessorkerne skaliert werden. Jeder Task bearbeitet dabei den ihm zugewiesenen Bereich der Datei (Parsing und Deserialization) und speichert die von den Anfragen benötigten Daten in geeigneten Datenstrukturen. Nicht benötigte Daten werden frühestmöglich verworfen.

8 Task-Scheduling

Die bisherigen Kapitel waren jeweils auf einzelne Aspekte und Aufgaben fokussiert, in diesem liegt der Fokus auf die Integration aller Teilaspekte in ein System. Alle Bausteine sollen möglichst effizient verknüpft werden um die Kerne des Systems möglichst gut auszulasten und so eine möglichst kurze Gesamtlaufzeit des Programms zu erreichen.

Bei der Parallelisierung des Systems sind verschiedene Varianten möglich. Ein klassischer Ansatz ist die *Inter-Query Parallelisierung*. Hier werden die einzelnen Aufgaben parallel zueinander ausgeführt. Bei Abhängigkeiten wäre eine einfache Lösung, die Ausführung

[2]Moderne Prozessoren müssen eine Vielzahl an zukünftig ausgeführten Instruktionen kennen um optimal ausgelastet zu sein (*pipelining*). Dazu wird der Pfad, den der Kontrollfluss bei einer Verzweigung nimmt, vom Prozessor erraten. Dies basiert auf einer Heuristik und ist daher nicht immer richtig, was dann zu einer teuren Invalidierung der CPU-Pipeline führt (*Branch-Miss*).

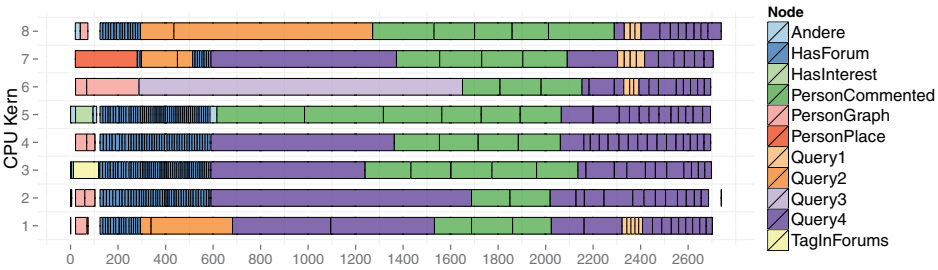


Abbildung 11: Ausführung der verschieden Arbeitspakete auf den CPU Kernen während der Programmlaufzeit.

der abhängigen Aufgaben durch eine globale Barriere von den vorhergehenden Anfragen zu trennen. Zur Verteilung der Aufgaben auf die Kerne können diese in eine zentrale synchronisierte Warteschlange eingereiht werden, Arbeiter die auf jedem der Kerne laufen überwachen diese Warteschlange und versuchen aus dieser immer Arbeit zu entnehmen, diese zu bearbeiten und dann das nächste Arbeitspaket zu holen. Durch die Synchronisierung der Warteschlange wird sichergestellt, dass eine Aufgabe jeweils immer nur einem Arbeiter zugewiesen wird. Durch diese dynamische Verteilung der Aufgaben ist sichergestellt, dass bis zur Verteilung der letzten Aufgabe aus der Warteschlange alle Kerne voll ausgelastet sind.

Zusätzlich zur parallelen Ausführung der verschiedenen Komponenten unseres Systems, haben wir die teueren Komponenten, wie zum Beispiel das Lesen der Dateien und die Ausführung von Anfragen vom Typ 4, zudem noch in sich parallelisiert. Diese *Intra-Query Parallelisierung* sorgt dafür, dass zum einen mehr Arbeitspakete zur Verfügung stehen und somit mehr Kerne genutzt werden können und aber auch dafür, dass es kein einzelnes großes Aufgabenpaket gibt, welches am Ende dann nur auf einem Kern ausgeführt wird, während die anderen Kerne warten müssen.

Mit diesen beiden Parallelisierungsvarianten kann bereits eine gute Auslastung der CPU-Kerne erreicht werden. An den globalen Barrieren gibt es jedoch einen Punkt, an dem die meisten Kerne warten müssen, bis auch der letzte Kern mit seiner Arbeit fertig ist und die Aufgaben hinter der Barriere begonnen werden können. Dies führt zu einem Verlust potenzieller Rechenleistung. Um dies zu vermeiden, haben wir die globalen Barriere entfernt und stattdessen die genauen Abhängigkeiten zwischen den verschiedenen Komponenten in einem Abhängigkeitsgraph modelliert. Für jede Komponente werden die Arbeitspakete erst an den Scheduler übergeben, sobald alle Aufgaben von vorhergehenden Komponenten abgearbeitet sind. Durch dieses Prinzip sind die Arbeiter bei der Bearbeitung der Arbeitspakete nie durch das Warten auf andere Arbeitspakete blockiert. Als weitere Optimierung unterstützt unser Scheduler die Priorisierung von Arbeitspaketen. Mit dieser Priorisierung ordnen wir die Ausführungsreihenfolge der zur Verfügung stehen Arbeitspakete so, dass Abhängigkeiten von Komponenten die viele Arbeitspakete generieren zuerst ausgeführt werden. Durch diese Priorisierung erreichen wir, dass schon früh möglichst viele Arbeitspakete zur Verfügung stehen, so dass kein Arbeiter dadurch blockiert ist, dass er keine Arbeit findet. Die Prioritäten wurden von uns anhand von Experimenten festgelegt.

Die Effektivität unserer Parallelisierung haben wir getestet, indem wir die Laufzeit unter Nutzung der verschiedenen Parallelisierungsmodelle für die Ausführung aller Wettbewerbsanfragen auf dem 100.000-Personen-Netzwerk gemessen haben. In Abbildung 10 ist sichtbar, wie die Gesamtlaufzeit durch zunehmend feinere Parallelisierung reduziert werden konnte. Abbildung 11 zeigt die zeitliche Ausführung der Komponenten auf den Kernen (markiert durch Farben). Dank der feingranularen Ausführung gibt es kaum Phasen in denen ein Kern nicht ausgelastet ist.

9 Zusammenfassung und Ausblick

In diesem Artikel wurde das Gewinnersystem des ACM SIGMOD Programming Contest 2014 beschrieben. Die Aufgabe des Programmierwettbewerbs bestand darin, ein hochperformantes Analysesystem für soziale Graph-Daten zu entwickeln. Nur durch die Kombination von algorithmischen Verbesserungen wie unserer Daten-parallelisierten Graph-Breitensuche, eigenen Heuristiken zur starken Einschränkung des Suchraums bei Top-k-Anfragen, feingranularem Task-Scheduling, und effizientem parallelisierten Laden von textuellen Rohdaten konnten wir das Mehrkern-System optimal ausnutzen und den Wettbewerb mit über 30% schnellerer Laufzeit als der Zweitplatzierte gewinnen.

Die präsentierten Techniken wurden bisher in einem eigenständigen System implementiert, können aber in bestehende Graph-Datenbanksysteme integriert werden. Wir arbeiten des Weiteren daran die Techniken in unser Hauptspeicher-Datenbanksystem HyPer [KN11] zu integrieren um Deep Analytics auf großen Graph-Daten effizient zu unterstützen.

Literatur

[APPB10] Virat Agarwal, Fabrizio Petrini, Davide Pasetto und David A. Bader. Scalable Graph Exploration on Multicore Processors. In *SC*, Seiten 1–11, 2010.

[BAP12] Scott Beamer, Krste Asanović und David Patterson. Direction-Optimizing Breadth-First Search. In *SC*, Seiten 12:1–12:10, 2012.

[BFG+13] Peter A. Boncz, Irini Fundulaki, Andrey Gubichev, Josep-Lluis Larriba-Pey und Thomas Neumann. The Linked Data Benchmark Council Project. *Datenbank-Spektrum*, 13(2):121–129, 2013.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7):422–426, Juli 1970.

[BM11] Aydin Buluç und Kamesh Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *SC*, Seiten 65:1–65:12, 2011.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest und Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd. Auflage, 2001.

[ES11] Stefan Edelkamp und Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[Flo14] FlockDB, 2014. https://github.com/twitter/flockdb.

[FNM85] Philippe Flajolet und G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *JCSS*, 31(2):182–209, 1985.

[HNH13] Stefan Heule, Marc Nunkesser und Alex Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In *EDBT*, 2013.

[Ior10] Borislav Iordanov. HyperGraphDB: A Generalized Graph Database. In *WAIM*, Seiten 25–36, 2010.

[JRW+14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande und Mike Stonebraker. VERTEXICA: Your Relational Friend for Graph Analytics! *PVLDB*, 7(13):1669–1672, 2014.

[KN11] A. Kemper und T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, Seiten 195–206, 2011.

[LBK14] Daniel Lemire, Leonid Boytsov und Nathan Kurz. SIMD Compression and the Intersection of Sorted Integers. *CoRR*, abs/1401.6399, 2014.

[LDB14] LDBC SNB: Generated CSV Files, 2014. https://github.com/ldbc/ldbc_snb_datagen/wiki/Generated-CSV-Files.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson und Jonathan Berry. Challenges in Parallel Graph Processing. *PPL*, 17(1):5–20, 2007.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser und Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*, Seiten 135–146, 2010.

[MBMMGV+07] Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez und Josep-L. Larriba-Pey. Dex: High-performance Exploration on Large Graphs for Information Retrieval. In *CIKM*, Seiten 573–582, 2007.

[MRS+13] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper und Thomas Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14):1702–1713, 2013.

[Neo14] Neo4j, 2014. http://neo4j.com/.

[Pcs14] SIGMOD 2014 Programming Contest Leaderboard, September 2014. http://www.cs.albany.edu/~sigmod14contest/.

[PKST11] Spiros Papadimitriou, U Kang, Jimeng Sun und Hanghang Tong. Centralities in Large Networks: Algorithms and Observations. In *SDM '11*, Seiten 119–130, 2011.

[SWL13] Bin Shao, Haixun Wang und Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, Seiten 505–516, 2013.

[TKC+14] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann und Huy T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *PVLDB*, 8(4):449–460, 2014.

[Tro05] Vadim Tropashko. Nested intervals tree encoding in SQL. *ACM SIGMOD Record*, 34(2):47–52, 2005.

Towards Workflow Benchmarking: Open Research Challenges

Cesare Pautasso
Vincenzo Ferme

Faculty of Informatics
University of Lugano (USI)
via Buffi 13
CH-6900 Lugano, Switzerland
c.pautasso@ieee.org
vincenzo.ferme@usi.ch

Dieter Roller
Frank Leymann
Marigianna Skouradaki

Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstr. 38
70569 Stuttgart, Germany
Dieter.H.Roller@iaas.uni-stuttgart.de
leymann@iaas.uni-stuttgart.de
marigianna.skouradaki@iaas.uni-stuttgart.de

Abstract: Workflow Management Systems (WfMS) provide platforms for delivering complex service-oriented applications that need to satisfy enterprise-grade quality of service requirements such as dependability and scalability. The performance of these applications largely depends on the performance of the WfMS supporting them. Comparing the performance of different WfMSs and optimizing their configuration requires that appropriate benchmarks are made available. In this position paper we make the case for benchmarking the performance of WfMSs that are compliant with the Business Process Model and Notation 2.0 standard and explore most of the challenges that one must tackle when constructing such benchmarks.

1 Introduction

Since the introduction of Web services [Pas05], workflow-based applications [LR97] have become the state-of-the-art programming model for integrated applications and information systems. They support the implementation of the two-level programming paradigm [WWC92], where the programming in the large is carried out using process models and the programming in the small is implemented via appropriate Web Services [ACKM04, GGKS02, WCL+05]. This approach has entered all business areas [KKL06, ZTP03] rang-

ing from simple forms-oriented applications, such as opening a new account in a bank or the approval of a travel plan in a software company, to complex applications, such as the planning for the construction of a car in the design department of a car manufacturer.

The performance of workflow-based applications depends basically on two factors: the performance of the workflow management middleware and the performance of the application components [LBK03]. The application components are either produced by the company running the application or have been provided by some software vendor. These are typically delivered as a service. We assume that these components have been designed and implemented using appropriate best practices and have been configured and tuned for optimal performance. It can be further assumed that the application components and the Workflow Management Systems (WfMS) are clearly separated; a situation that is normal if the WfMS is provided by some software vendor and/or implemented using the BPMN 2.0 standard [JE11]. In this case, the achievable overall performance of the application depends solely on the performance of the WfMS.

We find ourselves now at the same situation as database management systems in 1985, when Jim Gray published the first benchmark proposal [BBC+85]: agreement had been reached on an appropriate standard and the first set of database engines were delivered as commercials systems. Proposing a comparison benchmark posed an important challenge that sparked significant research activities in the field of databases resulting in an unexpected dramatic improvement of database technology; the performance of database engines measured through the TPC benchmark improved dramatically from 54 transactions/minute in 1992 to more than 10 million transactions/minute in 2010.

In the context of WfMSs, we have a similar situation: agreement has been reached on BPMN 2.0 as the standard for Business Process Management [Ley11] and more and more commercial and research WfMSs based on BPMN 2.0 are entering the market[1].

It is the right time to develop a benchmark for BPMN 2.0 workflow engines that helps to give an objective and quantitative comparison and as a result drives forward workflow middleware technology[2].

WfMSs belong to a different class of software systems [Wes07] as opposed to database management systems or programming language compilers, for which there are established benchmarks (e.g., TPC-C [Tra97], SPEC CPU [Sta11]) and well known performance analysis and optimization techniques [Jai91]. In contrast to databases, which passively react to a workload of queries [Gra92], WfMSs actively drive the orchestration of distributed services which lie outside of their control [LRS02]. As opposed to programming languages, which can be optimized for local execution, workflow languages make use of composition constructs which shift the focus of the optimization from local computations to distributed

[1] https://en.wikipedia.org/wiki/List_of_BPMN_2.0_Engines

[2] Business Process Management cannot be fully reduced to BPMN 2.0, as many other approaches (e.g., WS-BPEL, Case Management, Structured vs. Unstructured, Declarative/Rule-based vs. Imperative process modeling) have been proposed and are being pursued. This heterogeneity has been so far detrimental to the development of a comprehensive benchmark to compare systems that are very diverse in terms of the features they offer, the different types of middleware technology they leverage underneath and the meta-modeling paradigm they use to describe business processes. Thus, in the rest of this paper we assume a level playing field whereby the engines to be benchmarked comply with one standard.

interactions.

The objective of this position paper is to define the challenges involved in the construction of a solid benchmark for BPMN 2.0 based WfMSs that helps 1) analyze the performance behavior of WfMSs, 2) give a fair comparison to different systems and therefore, to 3) contribute to drive forward the progress of workflow technology.

The rest of this position paper is organized as follows: Section 2 presents work that is related to benchmarking WfMS and examines its relevance with respect to our goals. Section 3 discusses the main factors that contribute to determine the performance of a WfMS. Sections 4 and section 5 explore the logistic and technical challenges that designers of WfMS benchmarks must overcome. Section 6 discusses the relationship between the challenges and the performance factors, while Section 7 concludes this position paper.

2 Related Work

When it comes to evaluating the performance of complex systems such as modern service oriented architectures [GGKS02], there has been a lot of work focusing on each layer of the architecture [HZ06], going from the storage layer [Gra92, TZJW08, Cha95] all the way up through the middleware [BCM+05, SKBB09, GSC+04, PP08] into the Cloud [BKKL09]. In this section we present a brief summary of relevant benchmarking literature, including references to the most important surveys [WFP07].

Concerning WfMS, relatively less work can be found specifically targeting the topic of benchmarking their performance (e.g., [GMW00, BBD10a, DPZ11]). One of the first benchmarks that has been published was LabFlow-1 [BSR96]. The goal was to study the impact of the database on the workflow performance; in fact, the authors saw it more as a database benchmark rather than a workflow benchmark, comparing the overhead different database management systems used by a workflow engine. The benchmark was motivated by the statements made in [GHS95] that commercial WfMSs could not support applications with high-throughput workflows and would not meet the needs of the associated genome research center for high-performance workflows. Unfortunately, the structure of the workflows was not given, so that no judgment can be made how the benchmark would perform on a modern, state-of-the-art WfMS.

The next benchmark [GMW00], conducted in 2000 by the database group of Gerhard Weikum at the University of Saarland, was comparing the performance of a commercial WfMS with the one that has been developed by members of the group. The base was a rather simple e-commerce workflow that was described using state charts. The actual benchmark measured the throughput of each of the systems as well as the impact of the database work that was forwarded to a dedicated server. The maximum throughput of the systems was measured at 400 processes/hour running on a SUN Sparc 10. The results may have been good for the time of the benchmark; however, it can be assumed, that a state-of-the-art WfMS would significantly outperform such benchmark (in particular given today's hardware capabilities).

Hence, the need to introduce a benchmark that addresses the state-of-art features is widely

recognized [KKL06, WLR+09, RvdAH07, LMJ10]. As one could speculate the vendors of commercial systems are running internal benchmarks (e.g. [IBM11], [Act11], [IC07]) to come up with performance figures that they publish in their product documentation or provide to prospective customers upon special requests. However, there is still no commonly accepted way to compare the performance of different engines.

Towards this direction SPEC introduced in 2010 a subcommitee on SOA [Sta10] middleware benchmarking. The scope of the working group is rather broad, as it covers business process choreography among many other SOA middleware technologies.

More recently, SOABench [BBD10a] is a framework for the automatic generation, execution and analysis of test beds for evaluating the performance of service-oriented middleware. It has been used in [BBD10b] to compare the performance of several WS-BPEL engines (ActiveVOS, jBPM, and Apache ODE). The goal of the benchmark was to test the efficiency of the individual structural activities, such as `<sequence>`, `<flow>` with and without links, and `<while>`. Each activity type was used in a process model with five invoke activities, whose implementation was a simple servlet. The performance was compared using a different number of clients and different think times between subsequent requests. Unlike the work in [GMW00], the measurements focused on response time and identified several scalability limitations of the systems being tested.

A black-box approach to workflow benchmarking has been published in [DPZ11]. The goal of the authors is to compare the performance of five unnamed workflow engines for the XPDL language under various workload conditions. The main metric chosen is the completion time and the proposed reference processes only involve local computations (e.g., incrementing loop counters and generating random numbers). The outcome is that commercial engines outperform the freely available ones under heavy load conditions.

OpenESB [Sun07] and Din et al. [DES08] use a simple synthetic process in their benchmarks. The first focuses on load testing while the second one on stresses the response time behaviour. Roller [Rol13] and FACTS [LLHX10] perform load testing using one real-world process to stress an open source and a proprietary engine. Both of these works invoke external services through their processes. Silver [HHGR06] benchmarks two BPEL engines using 12 kernel processes. It performs a baseline test that measures the latency and the memory utilisation.

The challenges that are analysed in the sections 4 and 5, have as a goal to extend the aforementioned related work, with respect to the following key points: a) a workload mix of diverse complexity, and b) performance tests of different types that will consider a larger set of raw performance metrics and aggregate them into meaningful key performance indicators.

3 Performance Factors

The contents of a benchmark and the associated challenges for constructing such a benchmark can only be appreciated if one understands how the various aspects of a WfMS will impact the performance with which a WFMS can execute business processes (Figure 1).

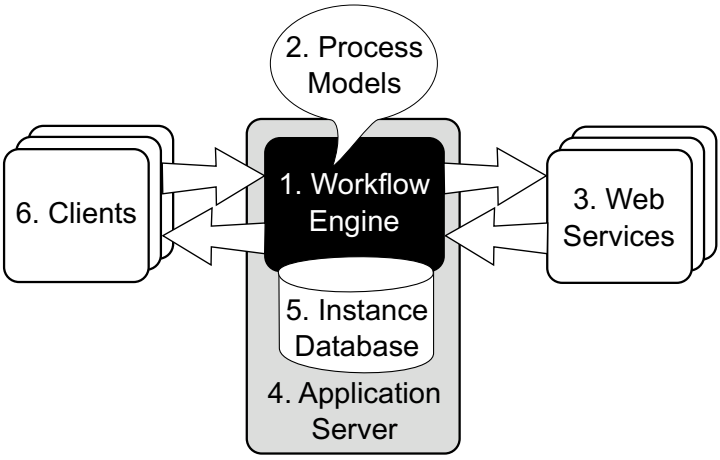


Figure 1: Factors affecting the performance of a WfMS

Six areas can be distinguished that contribute to the performance of a WFMS: (1) The architecture of the workflow engine, (2) The structure and complexity of the business processes that are being carried out, such as the level of parallelism or the handling of data, (3) The interactions that the business process carries out such as responding to a request from a client or invoking a Web Service, (4) The exploitation of the underlying middleware, such as transaction handling or message processing, (5) The management of data in a persistent store for the reliable execution of long-running processes, and (6) the load that the WfMS must sustain.

3.1 Workflow Engine Architecture

The internal architecture of the workflow engine is a very important factor contributing to the performance of the overall workflow-based application. Typically BPMN 2.0 engines will transform the process model into an internal representation that can be executed efficiently. In some cases, they may introduce several pools of worker threads to pipeline the concurrent execution of multiple, independent process instances and rely on the underlying application server middleware services to efficiently manage synchronous and asynchronous interactions. To achieve scalability, some engines have been designed to replicate key components to run across multiple processor cores, or even in a distributed environment such as a cluster of computers or a Cloud [PHA07]. There are also many possibilities concerning the performance optimizations that have been applied to modern workflow engines. These involve aspects such as:

• transaction management: where ACID properties need to be guaranteed both towards the external services participating in the workflow as well as within the engine, which is responsible for the consistent and dependable management of the state of each process

instance. Transaction boundaries can be adjusted to minimize conflicts, especially concerning short-lived micro-flows [HZ06].

• message processing: buffering, filtering, normalization, sorting, aggregation, and correlation of messages that are exchanged with the environment through the underlying middleware. Zero-copy or pass-by-reference techniques can be introduced as long as they do not violate the semantics of the process modeling language.

• failure handling and recovery: it is a common trade off to provide support for the persistent and reliable storage of the state of process instances by giving up some performance. This is particularly important for long running processes, which are likely to be affected by failures affecting the workflow execution engine itself. Write-through caching, persistent message queues and optimized logging representations can potentially make a big impact on the performance [LR98].

• monitoring: dynamically generating up-to-date reports on the internal activities of the system may require to combine OLAP characteristics within an architecture whose primary goal is to drive forward the execution of the active workflows, which is a form of OLTP [BGNS10]. Not only it is important to observe the impact of monitoring on the performance of the raw workflow engine, but also monitoring itself can be a source of specific benchmarking scenarios and challenges.

Having a benchmark would make it possible to fix the boundary conditions around the workflow engine (i.e., process models, interactions, workloads) so that it would be possible to measure and quantitatively compare the effect of critical design decisions on the engine's architecture.

3.2 Process Model Complexity

The complexity of a process model plays a significant role in the performance that can be achieved. It poses a significant challenge on the design and implementation to make sure that one does not pay for constructs if they are not defined in a business process. The complexity of a process model has many facets [Car07]; we illustrate here the most prominent ones which mostly affect the runtime performance of the process.

One important facet is the structure of the process model in terms of parallelism. Parallel processing within a business process can improve process execution time, in particular if time dependent processing, such as `timer` events, is available or other Web Services are invoked. However, extra execution resources are needed by the WfMS to cope with the parallel processing of each parallel path, including the safe handling of variables that are concurrently referenced in multiple paths.

Processing of data in transition conditions and assign activities is another important part of executing a business process. Depending on the size of the data stored within the involved variables and the level of sophistication of the XPath expressions, handling of data may require significant execution resources by the WfMS.

Compensation processing mandates that data and execution histories are kept so that pro-

cessing can be undone or alternate processing can be carried out. Storing persistent execution logs requires sufficient storage resources and also imposes an overhead during the execution of a transactional business process.

3.3 Interactions

The interactions that a business process carries out fall in two categories: the processing of client requests and the invocation of external Web Services. These are carried out using one of the following three interaction patterns: (1) synchronous request-reply, (2) asynchronous message-exchange, and (3) fire-and-forget. All patterns are defined via appropriate BPMN constructs (Figure 2) such as service task for (1), throw-catch of a message event or send-receive tasks for (2) and throw of a message event or send task for (3).

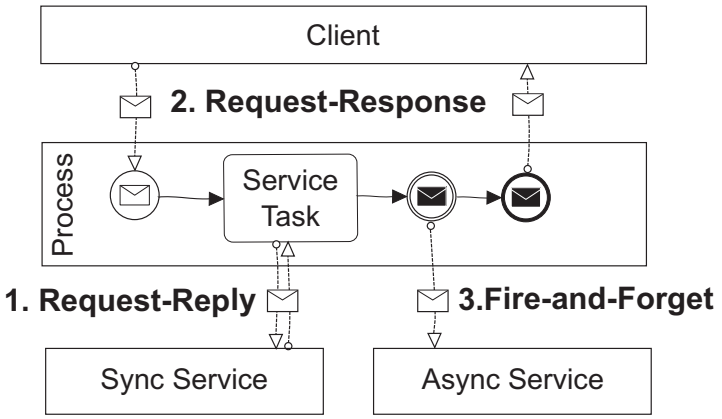


Figure 2: Interactions Handled by a WfMS

3.3.1 Request-Reply

The requestor starts the request-reply interaction pattern by calling the Web Service via the appropriate interface and then waits for the invoked Web Service to return the defined output value. When a client issues such a request, the WfMS loads the associated process model, creates a process instance, and has the appropriate receive handle the input value supplied in the request. Processing of the process instance continues by running the activities in the prescribed order, including possibly some interactions with other Web Services. The business process issues such requests, when a service task is defined. The WfMS in this case calls the defined Web Service and then waits until the response comes back. Finally, when an end state is reached, the associated variable is returned to the client. Even if

this pattern is designed for very short interaction in the range of seconds, the WfMS cannot make any assumptions about the response time of a called Web Service (when calling out) nor about the time does it take to execute a process instance (which determines for how long time the client remains connected to the engine waiting for a reply).

3.3.2 Request-Response

A requestor starts this pattern by calling the Web Service asynchronously; it is not waiting for any response, the response comes back to the requestor by having the invoked Web Service calling the requestor back at a specified end point reference. The requestor message contains correlation information that is returned by the Web Service along with its call back and that allows the requestor to correlate the response to the original request.

If the requestor is a client, the WfMS in the receive activity or in the catching message event extracts the correlation information and makes the extracted information part of the value returned in the response message. If a business process is the requestor, the Web Service is invoked via an appropriate send task or throwing a message event with correlation information being part of the input value.

Processing of the business process continues until a receive activity or an itermediate catching message event is encountered that has been defined with the same partner link. The WfMS now stops processing until a message is received for the receive activity or the catching message event. The correlation information in the message is used to locate the appropriate process instance (the one that issued the request).

As in the case of the request-reply, the WfMS can make no assumption about the time it takes for the called Web Service to respond. It must be able to process instantaneous responses (even within milliseconds) as well as responses that come after a very long time (e.g., more than a month).

3.3.3 Fire-and-Forget

This pattern is a simple asynchronous call, either issued by a client or by the business process. There is nothing special about this type of request, except that exception situations need very special treatments. For example, a business process invokes a Web Service which completes successfully, however the process aborted and may be automatically restarted since the processing of a WfMS is typically carried out within transactional brackets. From a performance perspective, these interactions are the ones which pose the least burden on the system, since they do not block and do not usually include correlation metadata.

3.4 Application Server Exploitation

WfMSs typically rely on functions that application servers provide, such as application scheduling, or timer services. The following functions are the most prominent ones from

a performance perspective.

Support of transactions All processing of a WfMS is carried out as transactions, so that a WfMS automatically provides forward recoverability; that means when the system fails, all changes applied to the business process are undone and the system is restarted with the original request. Note that backward recovery for business processes is provided via the compensation processing.

Queues Most WfMSs use message queues for interaction between the different components. These queues might become performance bottlenecks with concurent components subjected to heavy load.

Timer Services The timer event of the BPMN 2.0 specification as well internal functions of the WfMS exploit the timer services for starting appropriate actions at the right time.

Application management The various components of the WfMS are managed by the application server. This includes managing the execution lifecycle of the components that make up the WfMS, including dynamic loading and unloading of components.

3.5 Database Usage

Most business processes are long-running that means they cannot be carried out in a single execution step; those that do are typically called micro flows. This mandates that the WfMS stores the state of those process instances in some persistent store. The amount of data that must be stored thus becomes highly dependent on the execution time of process instances. The database is not only used by the WfMS internally but also to answer process queries, a facility that most WfMSs provide. In addition, most WfMSs provide audit trailing, a facility that writes an entry into a log table for each change in the state of a process instance. If audit trailing is activated, additional history data is generated and needs to be indexed and stored persistently. Extra storage is required if process instances must be, e.g., for legal reasons, kept for many years. Even with rapidly decreasing storage costs, the amount of storage required by different workflow engines is an important cost factor and may significantly change depending on the offered level of persistence.

3.6 Load/Request Management

A WfMS must sustain a plethora of different request scenarios [DES08], such as a large number of clients issuing moderately low-frequency requests, a small number of clients rapidly firing off many requests, or number of requests peaking at certain times during the day. Furthermore, users are expecting that WfMSs to some extent scale with the hardware that is available. Clustered or distributed workflow engine architectures which may be deployed on multicore servers on in the Cloud are starting to appear [ALMS09, LMJ10]. Such engine architectures may trade-off response time against throughput, making the

choice of workload very critical to give a fair assessment of their performance. Furthermore the chosen workload may be domain-specific [YÖK03], thus depend on the type of processes that are to be executed as part of the benchmark.

4 Logistic Challenges

This section introduces the set of challenges that one will find when trying to obtain information that correctly reflects the usage of WfMSs in the field.

4.1 Collecting Real World Scenarios

The notion of workflow-based applications [LR97] has entered all domains of applications, ranging from simple transaction-oriented processing in a bank, straight-through processing in financial institutions, the Enterprise Application Integration (EAI) scenarios of a production company, to the support of engineers in the design department of a car manufacturer. It is therefore necessary to collect as many real world scenarios as necessary to have a fair representation of the applications, which use WfMS as their base. Only then it is possible to come up with a benchmark that correctly reflects the usage of the WfMSs in the real world. The challenge here is to only collect as many scenarios as necessary but not more. It is very likely that a single benchmark can hardly reflect the diversity of applications or the different exploitation of workflow technology in the different industries, so one may end up with several benchmarks reflecting the level of sophistication of the WfMS and orthogonally to it several benchmarks for different industries. In this case, it may be required to collect more real world scenarios for the construction of the different benchmarks. It is understood that getting real world scenarios is also a challenge, since many companies do not want to disclose their business processes for competitive reasons.

4.2 Synthesizing Benchmark Flows

The underlying process models reflect the diversity of the applications, ranging from simple sequential processes with a few activities to complex, long running, highly parallel processes, from the invocation of simple synchronous Web Services to highly complex, context-based message exchange scenarios in order fulfillment processes. To keep the benchmark manageable, it is virtually impossible to construct a benchmark by just adding processes corresponding to each particular scenario for each usage segment. Thus one needs to extract the essence of each of the scenarios and construct a set of synthetic process models plus appropriate Web Services. The challenge is therefore to construct a minimal number of process models that correctly reflects the actual usage of the different process model constructs that the process model language offers. In other words, the set of process models making up the benchmark needs to reflect the usage pattern of the various

BPMN 2.0 constructs in real-world scenarios. We envisage a modular set of benchmarks including the mostly used BPMN 2.0 features in the core and the less frequently used features as optional extensions. For this purpose we intend to implement a static and dynamic analysis on the selected collection of process models. These analyses will export micro characteristics of process models such as which language elements are mostly used and to which extent, and also macro characteristics, which are namely reoccurring structures (e.g., fragments and patterns) in the collection of process models. This way we can automate the synthesis of representative processes.

4.3 General vs. Domain Specific Benchmark Flows

In a second step, the synthesized process models must be aggregated in such a way that they correctly reflect the appropriate usage patterns of the different scenarios; that means the execution of the different process models with their different process activities should reflect the sum of all scenarios, if possible. If that does not prove to be feasible, it is suggested to develop a set of benchmarks that reflect certain industries, such as banking or public sector. The cross-industry, function exploitation oriented approach has been also followed for the database benchmarks through the introduction of special benchmarks [PF00], such as the TPC-D [Tra95] benchmark for data warehouses, or the TPC-H [Tra] for decision support systems. Given the large number of application domains in which workflow technology has been successfully applied, it is challenging to select a suitable representative subset of domains from which to start from collecting and synthesizing domain specific benchmark process models.

4.4 Benchmark Number and Key Performance Indicators

The result of the benchmark should be preferably a single number (or as few values as possible). Only a single number allows the easy comparison of different WfMSs. Special care must be taken that the benchmark number is fair, that means that is not favoring particular architectures or implementations. Given the very large number of metrics that can be used in observing the performance of a WfMS [LBK03, WLR+09] (just to name a few examples: throughput, response time, resource and power consumption, network traffic consumed, peak number of concurrently active process instances, total number of started vs. completed process instances, SLA violations, failure rate, required memory/disk storage, recovery time) it will particularly challenging to select and aggregate them into a single meaningful number. Additionally, it will be necessary to choose between a black-box approach, relying on metrics which can be obtained non-intrusively, or a white-box approach which may require to instrument the workflow engine to observe its internal behavior in detail.

It is suggested that one follows a similar approach to what is currently done in database benchmarking where the obtained performance is provided as an absolute number (trans-

actions/minute) as well as cost-rated number (transactions/minute/euro). However, it is also important to investigate if response time would make a useful comparison metrics (or whether there is a strong correlation between response time and throughput).

5 Technical Challenges

This section presents a set of technical challenges that one will encounter during the design of a fair benchmark that takes into account specific features and characteristics of WFMS.

5.1 Application Implementation Impact Elimination

The database benchmarks are conceptually simple: a test client fires off a number of requests against the database management system. The test completes when all pre-defined request have been issued. This is quite different for benchmarking a WfMS where not only the workflow engine is involved but also the invoked Web Services contribute to the end-to-end performance; both require appropriate hardware resources (CPU, I/O, and Network). The challenge here is the elimination of these non-WFMS resources from the benchmark results so that only the WfMS proper is benchmarked. The typical approach of eliminating the application impact by short-cutting the invocation does not work for at least two reasons: (1) one has no access to the internal code of the engine for engineering the short cut, (2) the interaction between a process and its invoked Web Services is normally carried using conversation-based message exchange protocols.

5.2 Prevent System Overloading

The WfMS exposes its activities that receive messages from the outside as Web Services. Thus the interaction between the invoked Web Services and the process is via set of Web Service calls either invoked by the WfMS for the Web Services implementing the invoke activities in the process or by the invoked Web Service calling the Web Services that implement the receive activities of the process. The same is true in general for the clients requesting the execution of a process; the process either sends an asynchronous response back to the caller, or returns the result via a call back to the calling Web Service. Consequently a simple test client like the one that is constructed for database benchmarking cannot be used. The test client in database benchmarks just fires off requests and waits until the database management system comes back with an answer before submitting the next request. This notion of having the client carrying out synchronous calls will not work for driving the execution of processes in a WfMS. Thus the client can only fire off requests using some delay between the individual requests so that the WfMS is capable of handling all requests without overloading, thrashing or keeping requests buffered somewhere and eventually being throttled by the underlying messaging infrastructure. The challenge

would be to automatically adjust the delay rate so that the WfMS can cope with the requests.

5.3 Benchmarking Long Running Processes

The lifetime of processes ranges from milliseconds for a process that invokes another Web service to obtain, for example, the price of a particular book in a book store service to seconds for the handling of a sales transaction in straight-thru-processing, to weeks for an order process that handles the ordering of goods from a supplier, or even years for a process that controls the design process at a car manufacturer. The characteristics of those long running processes are twofold: (1) the instance database gets large and (2) neither the WfMS nor the database management system can take advantage of caching. The challenge is to find a method/setup that allows the benchmarking of those long-running processes without having to wait for years in order for the benchmark to complete.

5.4 System Internal Load Optimization

Some WfMSs exploit internal load optimization techniques to cope with the situation that the request load quite often varies with the time of the day. Thus they shift work to the time of the day of when the request load is lower. For example, IBM MQSeries Workflow postpones the resource-intensive deletion of process instances to lower-activity times. Likewise, Microsoft Windows Workflow Foundation can be configured not to always immediately move completed process instances to a specific database partition. The challenge is to accommodate this kind of adaptive/optimized behavior in the measurements for the throughput.

5.5 Correlation-based Message Exchange

Interactions between the invoked Web Services and the process are carried out as set of message exchanges. The messages are tagged with user-defined correlation information that helps the receiving WfMS and the invoked Web Services to find the appropriate process instance or other state information, in case the Web Service is not implemented as a process. Since the associated correlation information must be unique during the life time of a particular process instance, the challenge is to design and develop a test client in such a way that unique correlation tokens are generated when a request is sent to the WfMS for starting a process instance.

5.6 Performance Impact of Workflow Language Features

Workflow languages in general and BPMN 2.0 in particular provide a rich set of constructs to model iteration, concurrency and parallelism, exception handling and failure recovery, etc. During the design of the benchmark it will be important to consider whether all of the possible features of the workflow language will have to be covered by the test workloads, or only the ones that are actually and most frequently used in practice [MR08].

The challenge is to accommodate a variable degree of support for advanced language features by different engines. If the benchmark will require engines that provide full support for all language features, this may excessively restrict the number of engines that can be tested. This will enable the benchmark to be used as a tool to study the impact of given language features on the performance of a WfMS implementing (or not implementing) them. The hypothesis is that some engines may avoid implementing some features for performance reasons.

5.7 Reliability, Recovery and Robustness

An important feature of workflow engines concerns the guarantee of persistent execution for the process instances. If a failure occurs the state of the process instances is reloaded from persistent storage so that their execution can recover. The challenge is to define methods for controlled failure injection experiments in order to measure the failure rate of different systems as it is observed by the clients of the workflow engine. It should be noted that failure cannot be achieved through code modifications, but could be introduced by corrupting the messages as they reach the engine. At a different abstraction level, failures could also affect the services participating in the process [SPJ09], thus exercising the fault and compensation handlers specified as part of the benchmark processes [CKLW03]. Thus, it is important to be able to assess how well a given workflow engine can run processes over an unreliable execution environment but also run processes composing services of unreliable providers.

5.8 Monitoring

In addition to process execution, many workflow engines support the monitoring of the process execution by clients. In some case, *live monitoring* allows clients to track in real-time the progress of the process instances they are interested in. In other cases, it is possible to produce periodic monitoring reports, which give some statistics over the number of successfully or unsuccessfully completed process instances in a given time window. Depending on whether the reporting includes data from active processes or it mostly concerns a historical perspective on the instance database, the monitoring feature is likely to have an impact on the performance of workflow-based applications. The challenge is to take into account the impact of monitoring-related features on the process execution benchmarks.

Table 1: Summary: challenges and related performance factors

Logistic Challenges	Performance Factors: 1. Engine Architecture	2. Process Models	3. Web Service Interactions	4. Application Server	5. Instance Database	6. Client Workload
4.1. Collecting Real-World Scenarios		✓				
4.2. Synthesizing Benchmark Flows		✓				
4.3. General vs. Domain Specific Benchmark Flows		✓				
4.4. Benchmark Number and Key Performance Indicators	✓					✓
Technical Challenges						
5.1. Application Implementation Impact Elimination			✓			
5.2. Prevent System Overloading	✓			✓	✓	✓
5.3. Benchmarking Long Running Processes					✓	✓
5.4. System Internal Load Optimization	✓			✓		
5.5. Correlation-based Message Exchange	✓		✓		✓	✓
5.6. Performance Impact of Workflow Language Features	✓	✓				
5.7. Reliability, Recovery and Robustness	✓			✓	✓	
5.8. Monitoring	✓				✓	✓

Likewise, specific monitoring benchmarks can be proposed concerning the performance of the monitoring features of the engine, e.g., how well it scales with a large number of clients that are monitoring a large number of active process instances or retrieving historical reports on very large execution logs.

6 Discussion

The previous two sections present a potentially incomplete list of challenges that developers of a benchmark will face. Our list separates the challenges that are related to the collection of a representative set of usage scenarios (logistic challenges) from the technical challenges that are related to the specific characteristics of WfMS. Incidentally these are the same challenges the developers of WfMSs will face when developing functional

and stress tests of their systems (at least to some extent, since having access to the engine's code makes their life slightly easier). These challenges are also relevant when doing capacity planning to deploy a WfMS in production and enough resources need to be allocated to ensure an acceptable level of performance.

Table 1 summarizes the relationship between the challenges and the performance factors. For each performance factor, we analyzed whether the corresponding challenge has a potential impact on it. The analysis results are based on our experience with the design and architecture of workflow engines, with their performance optimization as well as with initial results of the BenchFlow project [SRF+15].

The logistic challenges mostly relate to finding representative and suitable process models to exercise all capabilities of the workflow engine being benchmarked in a way that is fair and representative of real-world usage scenarios. Attention must be also devoted to the design of a suitable workload model [AW96] that in our case corresponds to the chosen set of process models: lack of an appropriate and scalable workload models can make benchmarking experiments useless [MA01]. Obtaining a single benchmark number out of many possible Key Performance Indicators (KPIs) is influenced by what can be measured considering the engine architecture a black box subjected to a given workload from its clients.

The technical challenges are broadly related to all other performance factors. Eliminating the impact of the external application components and services that are integrated as part of a workflow will require to deal with the way the workflow engine interacts with its Web services. Bringing the system to its saturation point will require to apply a suitable client workload affecting all of its layers (from the database, through the application server middleware all the way to the workflow engine). The benchmarking results obtained with long running processes are likely to be affected by the instance database performance and will require to design specific client workloads. The definition of suitable correlation strategies for the benchmark is a cross-cutting challenge. Concerning the impact of workflow language features, these will need to be exercised by including them in suitable process models. Their performance will be affected by how efficiently they are supported by the workflow engine architecture. Including ways to measure the impact of the fault-tolerance and dependability aspects in the benchmark will need to take into account the effects of the database, the application server and the core of the workflow engine. Likewise, the result of a benchmark targeting the monitoring features will be affected by the organization of the instance database and require to define representative workloads of monitoring clients.

In our work we have begun to address some of these challenges, by making specific design decisions for the benchmark. Due to space limitations we cannot include many details in this paper. However at this point we would like to give a high level description of the approach that we plan to follow.

For challenge 5.1 we have contacted several companies and researchers to ask for process models witout focusing on a specific modelling language. In order to encourage the sharing of the models, we have suggested a) signing of confidentiality agreements and b) implementd a tool for obfuscating and anonymizing processes [SRPL14]. To face challenge 4.2 we analyze the collected models in terms of frequency of use of the BPMN 2.0

features, and frequent structures that are met in the models. With the appropriate usage of this information we plan to implement a workload generator, that will produce the workload according to WfMS specific features. This approach also satisfies challenge 4.3, and partially challenge 5.6 as we can adjust the workload to different features of a WfMS. It is planned that challenge 5.6 will also be addressed with compliance tests of BPMN 2.0 engines to the BPMN 2.0 standard. Challenges 3.2 and 3.3 will be addressed carefully designing a benchmark environment able to handle the diversity and the complexity of BPMN 2.0. Defining the KPIs (Challenge 4.4) will be approached by conducting multiple iterations of the benchmark. In each iteration more complex KPIs will be defined by aggregating metrics obtained with previous iterations and evaluating them based on the community feedback.

7 Conclusion

In this position paper we presented the case for a benchmark for WfMSs that helps compare the performance characteristics of WfMSs and therefore stimulates further research in this important middleware technology. Benchmarks should be simple, portable, scale to different workload sizes, and allow the objective comparisons of competing systems [Gra92]. Considering the large number of factors that impact the performance of a WfMS, the definition of a suitable benchmark remains a challenging open problem. The wide range of successful applications for workflow technology and also the complexity of workflow engines (which potentially interact with many different kinds of middleware) will warrant the exploration of novel approaches to the design of benchmarks specifically targeting the performance of such systems.

By presenting the set of open research challenges collected in this position paper, our goal is to start a discussion within the community interested in middleware for workflow and business process management on the need, potential benefits and possible design approaches for having a set of well-designed and widely accepted benchmarks for assessing, comparing and further improving the performance of WfMSs.

Acknowledgements

This work is partially supported by the Swiss National Science Foundation and the German Research Foundation with the BenchFlow - A Benchmark for WfMSs (DACH Grant Nr. 200021E-145062/1) project.

References

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, Applications*. Springer, 2004.

[Act11] Active Endpoints Inc. Assessing ActiveVOS Performance, 2011. `http://www.activevos.com/content/developers/technical_notes/`

assessing_activevos_performance.pdf.

[ALMS09] T. Anstett, F. Leymann, R. Mietzner, and S. Strauch. Towards BPEL in the Cloud: Exploiting Different Delivery Models for the Execution of Business Processes. SERVICES-2 '09, pages 670–677, July 2009.

[AW96] A. Avritzer and E. J. Weyuker. Deriving Workloads for Performance Testing. *Software: Practice and Experience*, 26(6):613–633, 1996.

[BBC+85] D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, et al. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, 1985.

[BBD10a] D. Bianculli, W. Binder, and M. L. Drago. Automated performance assessment for service-oriented middleware: a case study on BPEL engines. WWW '10, pages 141–150, 2010.

[BBD10b] D. Bianculli, W. Binder, and M. L. Drago. SOABench: Performance Evaluation of Service-oriented Middleware Made Easy. ICSE'10, pages 301–302, 2010.

[BCM+05] P. Brebner, E. Cecchet, J. Marguerite, P. Tůma, O. Ciuhandu, B. Dufour, L. Eeckhout, S. Frénot, A. S. Krishna, J. Murphy, et al. Middleware Benchmarking: Approaches, Results, Experiences. *Concurrency and Computation: Practice and Experience*, 17(15):1799–1805, 2005.

[BGNS10] L. Baresi, S. Guinea, O. Nano, and G. Spanoudakis. Comprehensive Monitoring of BPEL Processes. *IEEE Internet Computing*, 14(3):50–57, 2010.

[BKKL09] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. DBTest '09, pages 9:1–9:6, 2009.

[BSR96] A. J. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: A Database Benchmark for High-Throughput Workflow Management. EDBT '96, pages 463–478, 1996.

[Car07] J. Cardoso. Complexity analysis of BPEL web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

[Cha95] A. B. Chaudhri. An Annotated Bibliography of Benchmarks for Object Databases. *SIGMOD Rec.*, 24(1):50–57, 1995.

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana. Exception Handling in the BPEL4WS Language. BPM '03, pages 276–290, Eindhoven, The Netherlands, 2003.

[DES08] G. Din, K.-P. Eckert, and I. Schieferdecker. A Workload Model for Benchmarking BPEL Engines. ICSTW '08, pages 356–360. IEEE, 2008.

[DPZ11] F. Daniel, G. Pozzi, and Y. Zhang. Workflow Engine Performance Evaluation by a Black-Box Approach. ICIEIS '11, pages 189–203. Springer, November 2011.

[GGKS02] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153, 1995.

[GMW00] M. Gillmann, R. Mindermann, and G. Weikum. Benchmarking and Configuration of Workflow Management Systems. CoopIS '00, pages 186–197, 2000.

[Gra92] J. Gray. *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 2nd edition, 1992.

[GSC+04] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. v. Engelen, and M. J. Lewis. Toward Characterizing the Performance of SOAP Toolkits. GRID '04, pages 365–372, 2004.

[HHGR06] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman. Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices. ICSOC '06, pages 503–508. Springer, 2006.

[HZ06] C. Hentrich and U. Zdun. Patterns for Business Object Model Integration in Process-driven and Service-oriented Architectures. PLoP '06, pages 23:1–23:14, 2006.

[IBM11] IBM SAP. SAP NetWeaver Business Process Management Performance, Scalability, and Stability Proof of Concept. Technical report, IBM SAP International Competence Center (ISICC), Walldorf, Germany, 2011.

[IC07] Intel and Cape Clear. BPEL Scalability and Performance Testing. White paper, 2007.

[Jai91] R. Jain. *Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[JE11] D. Jordan and J. Evdemon. Business Process Model And Notation (BPMN) Version 2.0. Object Management Group, Inc, January 2011.

[KKL06] R. Khalaf, A. Keller, and F. Leymann. Business processes for Web Services: Principles and applications. *IBM Systems Journal*, 45(2):425–446, 2006.

[LBK03] T.-K. Liu, A. Behroozi, and S. Kumaran. A performance model for a business process integration middleware. CEC 2003, pages 191–198, 2003.

[Ley11] F. Leymann. BPEL vs. BPMN 2.0: Should You Care? volume 67 of *BPMN '10*, pages 8–13. Springer, 2011.

[LLHX10] A. Liu, Q. Li, L. Huang, and M. Xiao. Facts: A Framework for Fault-Tolerant Composition of Transactional Web Services. *IEEE Transactions on Services Computing*, 3(1):46–59, 2010.

[LMJ10] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web*, 4(1):2:1–2:33, January 2010.

[LR97] F. Leymann and D. Roller. Workflow-Based Applications. *IBM Systems Journal*, 36(1):102–123, 1997.

[LR98] F. Leymann and D. Roller. Building a robust workflow management system with persistent queues and stored procedures. ICDE '98, pages 254–258, 1998.

[LRS02] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[MA01] D. A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall, 1st edition, 2001.

[MR08] M. Muehlen and J. Recker. How much language is enough? Theoretical and practical use of the business process modeling notation. CAiSE 2008, pages 465–479. Springer, Springer Berlin Heidelberg, 2008.

[Pas05] J. Pasley. How BPEL and SOA Are Changing Web Services Development. *IEEE Internet Computing*, 9(3):60–67, May 2005.

[PF00] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Rec.*, 29(4):64–71, December 2000.

[PHA07] C. Pautasso, T. Heinis, and G. Alonso. Autonomic resource provisioning for software business processes. *Information and Software Technology*, 49:65–80, January 2007.

[PP08] P. Pääkkönen and D. Pakkala. Benchmark of middleware protocols for application and service interaction. MUM '08, pages 40–47, 2008.

[Rol13] D. H. Roller. *Throughput Improvements for BPEL Engines: Implementation Techniques and Measurements applied in SWoM*. PhD thesis, University of Stuttgart, 2013.

[RvdAH07] N. Russell, W. M. van der Aalst, and A. Hofstede. All That Glitters Is Not Gold: Selecting the Right Tool for Your BPM Needs. *Cutter IT Journal*, 20(11):31–38, 2007.

[SKBB09] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, 2009.

[SPJ09] S. Stein, T. R. Payne, and N. R. Jennings. Flexible provisioning of web service workflows. *ACM Transactions on Internet Technology*, 9(1):2:1–2:45, February 2009.

[SRF+15] M. Skouradaki, D. H. Roller, L. Frank, V. Ferme, and C. Pautasso. On the Road to Benchmarking BPMN 2.0 Workflow Engines. ICPE '15, 2015.

[SRPL14] M. Skouradaki, D. Roller, C. Pautasso, and F. Leymann. BPELanon: Anonymizing BPEL Processes. ZEUS '14, pages 9–15, 2014.

[Sta10] Standard Performance Evaluation Corporation. SPEC SOA Subcommittee, February 2010. `http://www.spec.org/soa/`.

[Sta11] Standard Performance Evaluation Corporation. SPEC CPU2006 Version 1.2, September 2011.

[Sun07] Sun Microsystems. Benchmarking BPEL Service Engine, 2007. `http://wiki.open-esb.java.net/Wiki.jsp?page=BpelPerformance.html`.

[Tra] Transaction Processing Performance Council. TPC-H. `http://www.tpc.org/tpch/`.

[Tra95] Transaction Processing Performance Council. TPC-D, 1995. `http://www.tpc.org/tpcd/`.

[Tra97] Transaction Processing Council (TPC). TPC Benchmark C (Online Transaction Processing Benchmark) Version 5.11, February 1997.

[TZJW08] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A Nine Year Study of File System and Storage Benchmarking. *Transaction Storage*, 4(2):5:1–5:56, 2008.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture*. Prentice Hall, March 2005.

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, and Architectures*. Springer, November 2007.

[WFP07] M. Woodside, G. Franks, and D. Petriu. The Future of Software Performance Engineering. FOSE '07, pages 171–187, 2007.

[WLR+09] B. Wetzstein, P. Leitner, F. Rosenberg, I. Brandic, S. Dustdar, and F. Leymann. Monitoring and Analyzing Influential Factors of Business Process Performance. EDOC '09, pages 141–150, 2009.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming: A Paradigm for Component-Based Programming. *Communications of the ACM*, 35(11):89–99, 1992.

[YÖK03] B. B. Yao, M. T. Özsu, and J. Keenleyside. Xbench-a family of benchmarks for XML DBMSs. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*, pages 162–164. Springer, 2003.

[ZTP03] O. Zimmerman, M. Tomlinson, and S. Peuser. *Perspectives on Web Services: Applying SOAP, WSDL, and UDDI to Real-World Projects*. Springer, September 2003.

Who Watches the Watchmen?
On the Lack of Validation in NoSQL Benchmarking

Wolfram Wingerath, Steffen Friedrich, Felix Gessert, Norbert Ritter

Databases and Information Systems Group
University of Hamburg
{wingerath, friedrich, gessert, ritter}@informatik.uni-hamburg.de

Abstract: There are numerous approaches towards quantifying the performance of NoSQL datastores with respect to dimensions that are notoriously hard to capture such as staleness or consistency in general. Many of these approaches, though, are built on assumptions regarding the underlying infrastructure or the test scenario and may lead to invalid results, if those assumptions do not hold. As a consequence, in-depth knowledge of both the system under test and the benchmarking procedure is required to prevent misleading results. In this paper, we want to make the case for more experimental validation in NoSQL benchmarking to uncover the bounds of existing benchmarking approaches.

1 Introduction

Relational databases (SQL databases) were once considered to be the one solution for all data storage and management problems. Basically, they all share the same query language and have very similar properties and performance characteristics. Everyone has a rather clear picture of what a relational database can do and what the implications of the different ACID flavours are (e.g. in terms of possible anomalies). Since many performance characteristics of SQL databases are dictated by their design, a reasonable performance evaluation of different SQL databases can be conducted by comparing their throughput and request latencies under a given workload for all supported ACID isolation levels. In contrast to the rather uniform SQL landscape, though, many different architectures with very different properties and performance characteristics are subsumed under the term **NoSQL**. In 2010, Yahoo! published the Yahoo! Cloud Serving Benchmark (YCSB) [CS+10] as the first noteworthy attempt at a benchmark across different NoSQL systems.[1] However, the YCSB only captures raw performance in terms of request throughput and latency under simple CRUD operation mixes (insert, read/scan, update, delete) and neglects all other criteria that describe the performance of distributed database systems. While measuring throughput and latency may have been sufficient to support a conclusive performance comparison between SQL databases, it is by far not enough to do the same in the field of NoSQL databases. During normal (i.e. fault-free) operation, throughput and latency are key to every NoSQL database, but the degree to which data are consistent (are there

[1] Several benchmarks had already been available before YCSB, but they were mostly built for performance verification throughout development and were not widely used to compare different systems.

stale reads? How stale are they?) and whether as well as in which scope transactions are available (single-key? Multi-key? Only across keys in the same data partitions?) cannot be ignored. Unlike traditional relational databases, many NoSQL systems are explicitly designed to withstand and even automatically recover from single-node outages, different kinds of network partitions or other failures that are guaranteed to happen in a distributed environment. Therefore, information on the impact that these anomalies have on consistency, durability, read and write availability or more generally request throughput and latency are crucial for evaluating NoSQL database system performance. In [FW+14], we already surveyed current efforts on the quantification of these performance characteristics (and staleness in particular) which are notoriously hard to capture and we concluded that many approaches rely on implicit assumptions that do not hold in practice, e.g. perfect clock synchronisation between different physical machines or low network communication delays. However, it is hard to evaluate the severity of such conceptual flaws from theory alone: just like a flawless benchmarking concept may lead to wrong measurements when implemented badly, a measurement scheme that allows potentially unbounded imprecision may yield useful results in practice.
In this paper, we argue that not only the concept behind a measurement scheme should be validated in order to explore the boundaries within which valid results can be obtained, but the actual implementation should be validated as well. In particular, we make the following contributions: we motivate practical benchmarking validation by illustrating weaknesses in existing NoSQL benchmarks (Section 2) and describe **SickStore** (Section 3), a single-node datastore we developed for validation that behaves like a distributed system and can simulate anomalies such as staleness. As a demonstration of SickStore's usefulness, we further conduct an experimental validation of staleness values obtained with YCSB++ (Section 4). Section 5 concludes the paper.

2 Why Validate Benchmarking Tools?

There are several approaches towards benchmarking staleness in distributed database systems that conduct their measurements in such a way that there are no guarantees on precision. In [FW+14], we described the ideas behind a number of database benchmarks and pointed out several weak spots in their design, concentrating on work related to staleness measurements. In this section, we first present our notion of staleness and then provide more detailed insights into the two staleness benchmarks that are most sophisticated in our opinion. Our goal is to convince the reader that benchmark validation is important as there are various possible sources of arbitrary imprecision, for example clock drift, network latency or mere software faults.
What is Staleness, Anyway? A stale read is a read operation that returns an outdated object version. When a data object is written to an asynchronously replicated datastore, there is a certain timeframe during which the different replicas of this data item diverge, because the write is visible on some, but not all replicas. This timeframe represents the actual data-centric staleness window, because stale reads of the written object may be served as long as the write operation has not been applied to all replicas. Data-centric staleness cannot be measured easily, though, because it requires knowledge of the internal system state and therefore cannot be implemented in a system-agnostic and generic way. Furthermore, the

staleness window that is actually observable on the client-side during a given workload might be sufficient for many use cases or even more relevant than the exact point in time at which data in the system become consistent. Therefore, virtually all existing staleness benchmarks measure the client-centric staleness, i.e. they try to capture how long after a write operation the system keeps serving stale data.

Bermbach et al. Staleness Measurement. Bermbach et al. [BT14] create a global log file from multiple distributed servers under the assumption of perfect clock synchronisation. As illustrated in Figure 1, the basic setup comprises the datastore under test, one server that runs a common YCSB workload and one writer and several readers for the staleness measurement.[2] To determine the staleness window of a write operation, the writer periodically writes its local time and a version number to the datastore, while the readers repeatedly poll the written data item and log their own local time for each received value. The staleness window can eventually be computed as the difference between the timestamp of the last stale read (local time of a reader) and the timestamp of the initial write (local time of the writer). The timestamps that are used for the computation of the staleness window are highlighted in the illustration: the writer updates the version number at timestamp 3 and the two readers do not see the updated version before timestamps 8 (Reader 1) and 7 (Reader 2), so that the staleness window is computed as $8 - 3 = 5$. Obviously, this procedure only delivers an approximation of the staleness window whose precision depends on how well the local clocks of the writer and the readers are synchronised. For example, only the last read timestamp of Reader 1 is used for the computation, since only the *last* stale read is relevant which (under the assumption of synchronised local clocks) comes from Reader 1. Accordingly, if the writer and Reader 1 are perfectly in sync, while Reader 2 lags 10 time units behind, only the last stale read from Reader 2 taken into account and the measured staleness window becomes $16 - 3 = 13$. If the writer's local clock lags behind or if one of the reader's local clock is significantly faster, it may even become negative. Clock drift has already been identified as a possible cause for dubious experimental results

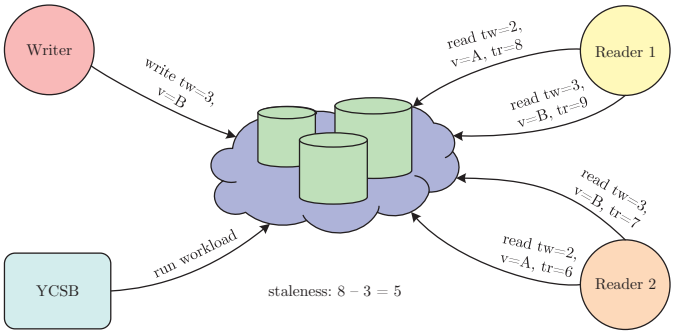


Figure 1: The basic set up of the staleness benchmark by Bermbach et al. [BT14].

by Bermbach et al. themselves [BZS14]. But even in a well-synchronised environment, this approach still does not produce exact results, because the timestamp that is written

[2]Writer and reader are placed on separate machines for several reasons: first, they might experience reduced or no staleness at all when placed on the same machine, e.g. when their requests are routed to the same replicas. Second, a single machine might not be able to saturate the system under test. Third, real-life clients are also (geographically) distributed.

to the datastore represents the point in time directly *before the write request was sent*, whereas the point in time *directly after the write request was acknowledged* is when data actually become stale. Depending on the required precision, the network latency and the time the system needs to process the write request, the staleness measurement may or may not be a reasonable approximation of the actual staleness window. Apart from that, the read/write ratio of the YCSB workload may be significantly disrupted by the additional reads and writes during the staleness measurement.

YCSB++. Another example of a staleness benchmark that gives rise to potentially unbounded measurement error[3] is the YCSB++ [PP$^+$11] which, in contrast to the approach by Bermbach et al., does not rely on clock synchronisation, but instead works with notification between writer and reader[4]. In principle, the writer inserts[5] new objects and enqueues their keys to a queue administered by a ZooKeeper service on a dedicated server, while the reader repeatedly dequeues, tries to read the corresponding values and eventually approximates the staleness window as the lag between the initial receipt of a key and the retrieval of the inserted value. Figure 2 shows the individual steps of a YCSB++ staleness measurement with one writer and one reader in a simplified way. The individual steps of the experiment are demarcated as i_n where i represents *the order* in which all steps happen and n represents *where* (at which node) each step happens (r for reader, w for writer, s for store). For illustration purposes, we use these demarcations as though they were (global) timestamps if appropriate. Further, we assume write operations to be applied at the moment of acknowledgement and we assume read operations to take place instantaneously when they are received. Initially, both writer and reader are given the signal to start the experiment by ZooKeeper (received at 1_w and 1_r, respectively). The reader tries to dequeue (2_r) the first element of the ZooKeeper queue, but has to wait, since the queue is empty. Meanwhile, the writer issues (3_w) the request to insert object x. The datastore receives the request (4_s), processes it and finally sends (5_s) the acknowledgement back to the writer. When the writer has received the acknowledgement (6_w) and enqueues (7_w) x to the ZooKeeper queue. The reader is notified of the new key in the queue, dequeues it and receives it (8_r). Object x. According to the basic idea as we also described it in our survey [FW$^+$14], the reader repeatedly requests object x, until x is actually returned. In more detail, though, the reader communicates with the ZooKeeper service after every unsuccessful read attempt: After the first read request (issued at 9_r, received at 10_s) returns null (sent back at 11_s, received at 12_r), the reader pushes x back to the ZooKeeper queue, removes the next item in the queue and continues the procedure with this item. When the reader eventually removes x again from the queue (13_r), it starts another read attempt (issued at 14_r, received at 15_s, answered at 16_s) and finally receives (17_r) the inserted object x. The staleness window is then approximated it as $17_r - 8_r$, i.e. the time between the reader receiving a key and the reader successfully retrieving the corresponding value. The *data-centric staleness window* starts at 5_s and ends somewhere after 10_s and before 15_s. The *client-centric staleness window* is $10_s - 5_s$, but it cannot be computed, because

[3] Our initial understanding of the YCSB++ approach was that it provided a lower bound for the staleness window as we reported in our survey on NoSQL OLTP benchmarking [FW$^+$14]. We have to admit, though, that that this was not the case and that YCSB++ does not provide any bounds on precision during staleness measurements.

[4] In the YCSB++ paper, writer and reader are referred to as **producer and consumer**.

[5] Note that staleness measurement are only possible for inserts and *not* for updates.

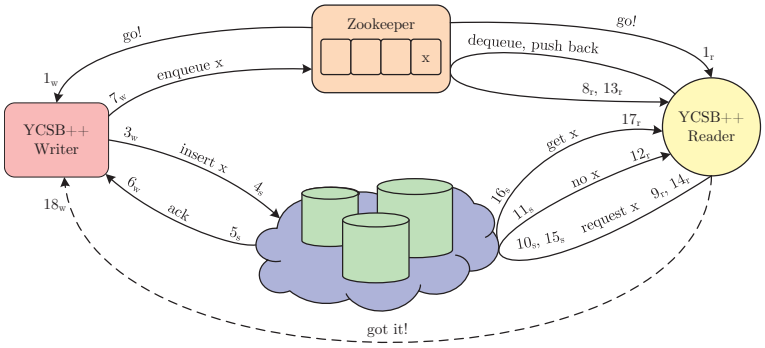


Figure 2: The basic experimental setup for staleness measurement in YCSB++.

all timestamps are taken from client machines and therefore neither 10_s nor 5_s are available. Further, the difference between 10_s and 5_s would also be distorted by clock drift, even if they were available, because the insert and the read request are not necessarily processed on the same datastore node: if the exact same machine that acknowledged the insert also responded to the read request, most systems would not exhibit any staleness whatsoever, since x would be visible directly after the acknowledgement. Hence, in case of an actual stale read, the insert and read request may be assumed to have been processed by two distinct machines that do not share the same local clock. But even though the exact client-centric staleness window cannot be obtained, it is possible to completely rule out the possibility of clock drift by using only timestamps from either the writer or the reader to approximate the staleness window: since $5_s < 8_r < 9_r < 10_s$, a *lower bound* for the staleness window can be computed as $9_r - 8_r$.[6] Similarly, an *upper bound* could be computed, if the reader reported back to the writer after having read x. Assuming the writer received such a notification from the reader at timestamp 18_w, an upper bound could be computed as $18_w - 3_w$, since $3_w < 5_s < 10_s < 18_w$ holds. Since the staleness window measured by YCSB++ does not contain the actual staleness window $(10_s - 5_s)$ and is not contained in it, it is neither a lower nor an upper bound: depending on the network delay and the time that an item remains in the ZooKeeper queue, there may be significant lag between the writer enqueueing the key and the reader initially receiving it (**read-after-write lag**) as well as between the reader pushing the key back into the queue after an unsuccessful read attempt and receiving it again. The former kind of lag may prevent the reader from observing any staleness, simply because it is not notified soon enough, while the latter may lead to arbitrarily large staleness measurements, because the reader takes too long to start another read attempt. YCSB++ avoids imprecision through clock drift, but allows additional delays through network commutation and the use of a queue which might not be delay between the reader and the ZooKeeper coordination service. In addition and similar to the experiments by Bermbach et al., it is unclear in what way the staleness measurements influence the read/write ratio of the YCSB workload that is run by the writer.

[6]Note that this lower bound can be very close to zero, if the read is successful at first or second attempt. If more than two read attempts are required, though, the lower bound can assume values significantly greater than zero.

3 Designing for Single-Node Inconsistency: SickStore

The major difficulty in validating existing staleness benchmarks is that none of the timestamps that would be required to determine server-side (data-centric) staleness are actually available for measurement, either because they would require internal system knowledge or because the database system itself is distributed and therefore subject to clock drift. In order to have a gold standard that can be used to evaluate the quality of staleness benchmarking results, we developed a **s**ingle-node **inc**onsistent **k**ey-value **store**: **SickStore**. SickStore is a single-node key-value store that is able to simulate the behaviour of a distributed key-value store. It is thus able to provide the same anomalies as a distributed system, but at the same time consistent knowledge about the system state at any point in time. Its main building blocks are the internal backend that holds all data, several virtual server nodes that listen on different ports of the same machine and a processing layer in between that does the actual request processing and decides what data can be served from what server node. Figure 3 shows an example where one write request and two read requests are executed against a SickStore instance with three virtual servers A, B and C. A write request to insert or update value "Bob" under key x arrives at timestamp 4 at server A (1) which forwards the request including the key, the value and the write timestamp $tw = 4$ to the query handler (2). The query handler requests the staleness generator to generate a staleness window for each virtual server (3) and receives one in form of a map that associates each server with its individual staleness window in ms. The query handler has the backend store everything (value, write timestamp, staleness associations) under key x (5). The backend maintains a list of all writes that have occurred under every key in order of their arrival at the backend. The acknowledgement goes back to the server that received the request and is then returned to the client.

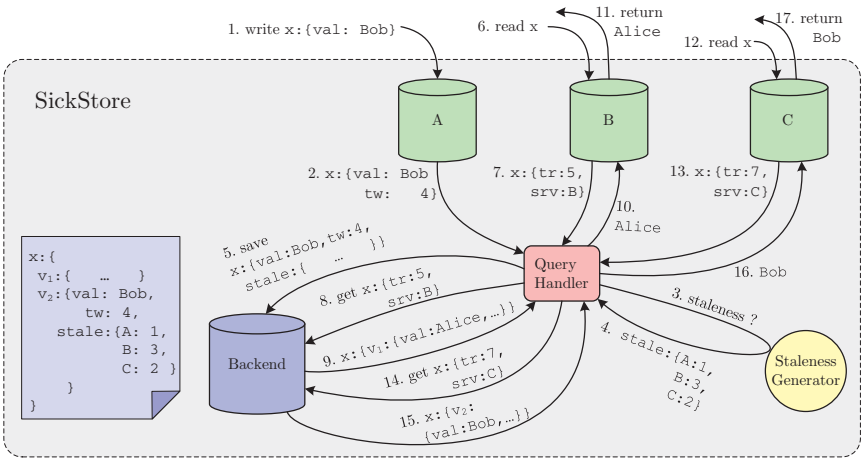


Figure 3: The basic architecture of SickStore: the different virtual server nodes accept and reply to all requests, but they all display a certain staleness, while the central versioned key-value store (backend) keeps track of all changes and is aware of the true system state at any time.

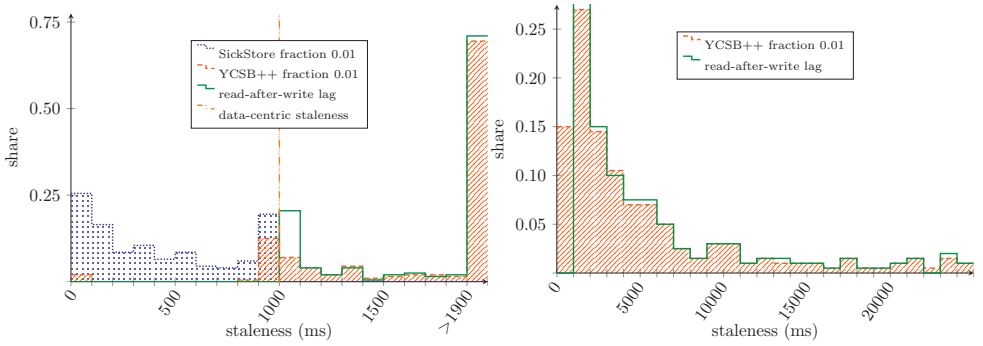
Shortly thereafter at timestamp 5, a read request for x arrives at server B (6). Similar to the request discussed above, this read request is also forwarded to the query handler together with the read timestamp $tr = 5$ and the name of the server node that received the request (7). The query handler then directly sends these information to the backend (8) and requests the most recent version of x that server B is allowed to serve; since the most recent version has been written at timestamp 4, but has a staleness window of 3 associated with server B, it is not visible for server B before timestamp $4 + 3 = 7$. Since the current read timestamp is 5, the query handler receives (9) the value of the most recent version that *actually is* visible for server B, in this example the value "Alice", and hands it to server B (10) which then returns it to the client (11). Another read request for the same key arrives at server C at timestamp 7 (12) and is again forwarded to the query handler (13) together with the read timestamp $tr = 7$ and the name of the server C. Again, the query handler requests the most recent visible version, but actually receives the most recent version this time, because the read request arrived at timestamp 7 and the C-associated staleness window is 2 (i.e. because $7 \geq 4 + 2$). Naturally, the value is eventually returned to the client. Since all SickStore components have the same local clock time and since there is virtually no communication delay between them, the processing layer can precisely implement a pre-defined visibility delay for each virtual server and thus make sure the data are served exactly as stale as they are supposed to be served. Further, the modular design of SickStore allows to interchange different implementations of the staleness generator, so that our constant staleness generator can easily be replaced by a more sophisticated one in the future. Depending on what criteria are used to determine the staleness that is exhibited, arbitrary replication or sharding strategies in arbitrary network topologies can thus be simulated.

4 Experimentally Validating Latency Measurements With YCSB++

The benchmarks discussed in Section 2 (and presumably many more that we do not discuss here) bear the potential to display arbitrary measurement error without the user noticing. As a consequence, their precision and actual meaningfulness are hard to evaluate by merely speculating about the impact of workload disruption, clock drift (Bermbach et al.), communication delay (YCSB++) or other possible ramifications in a given experimental setup. We claim that a performance evaluation using these benchmarks is not possible without experimental evidence regarding whether, under what circumstances and to what extent these benchmarks are actually useful. In [BZS14, p. 42], Bermbach et al. already concluded that clock drift during their measurements has severe implications on the measured staleness and thus can easily lead to misleading results. In this section, we show that YCSB++ is also prone to severe measurement errors, even though clock drift causes none of them.

Experimental Setup. All experiments were executed in a virtualised environment hosted on Xeon servers with 2.1 GHz 6-core CPUs: the YCSB++ writer and reader each had two VCPUs and 4 GB of memory, the ZooKeeper service and SickStore each had a single VCPU and 2 GB and 30 GB of memory, respectively. In principle, the YCSB writer is capable of executing an ordinary YCSB workload against the datastore under test and submits only a small *fraction* of the insert operations (1% by default) to ZooKeeper for staleness measurements. To keep the fraction scalable in the range between 0% and 100%,

we decided to have the YCSB++ writer run an insert-only workload and thus avoid any operations whose staleness could not be measured.[7] To make sure that neither SickStore nor the YCSB++ writer would not become bottlenecks during the experiments, we determined their maximal throughput, first: our SickStore deployment achieved consistently more than 3000 operations per second under insert-only, read-only and mixed insert-read (50/50) workloads using single-threaded vanilla YCSB and also under an insert-only workload using a single-threaded YCSB++ writer with staleness measurement disabled[8]. To further prevent any dependencies between individual experiments, we shutdown SickStore's JVM after each experiment and started a new instance for each experiment. We configured SickStore to exhibit *constant staleness of exactly 1000 ms*.



(a) The YCSB++ staleness measurements do not reflect the actual data-centric well.

(b) The measured staleness distribution has a long tail and many outliers.

Figure 4: Target throughput 500 ops/sec, fraction 1%, 1000 ms data-centric staleness.

Measuring Staleness. We started our first experiment with a fraction of 1% as proposed in the YCSB++ paper and we only used a target throughput of 500 inserts per second which translated to 5 inserts per second with subsequent staleness measurement. Both writer and reader were single-threaded the experiment stopped after 200 staleness measurement. The histogram in Subfigure 4b illustrates the frequency of actual client-observable staleness (dotted line), the staleness windows measured by YCSB++ (dashed line), the constant data-centric staleness (dotted line) and the read-after-write lag measured at SickStore (solid line), i.e. the time between the initial write and the last (successful) read operation. Since YCSB++ tries to measure the actual data-centric staleness as an approximation of the read-after-write lag and since the actual data-centric staleness window exhibited by SickStore was exactly 1000 ms, one would expect the measured staleness windows and also the read-after-write lag to be close to 1.000 ms. As illustrated in the plot, though, the experimental results did not correspond to our expectations at all. First and foremost, less than 25% of all measured staleness values (47 out of 200) are between 900 and 1000 ms, 4 operations were not observed as stale at all and more than half of all operations (119) displayed 1900 ms of staleness or more.

The histogram of all values in Subfigure 4b shows that some staleness values reported by

[7] As described in Section 2, YCSB++ Only measures staleness values for inserts and not for update operations.

[8] We set the fraction of staleness measurements to 0, but also started a single-threaded consumer and a ZooKeeper and had them perform their usual barrier synchronisation.

YCSB++ exceed the actually exhibited staleness by more than the 20-fold. This behaviour can be explained with the frequency distribution of the read-after-write lag which appears to dominate most staleness measurements. Since the read-after-write lag represents the time that passes between the initial write and successful read, it also indicates the time that passes between the last unsuccessful read and the final successful read, because any read that is received by SickStore more than 1000 ms after the initial write would automatically result in a success and thus would finalise the read-after-write lag for that value. Therefore, the time that lies between the last two read operations of a measurement can at most be 1000 ms smaller than the read-after-write lag: if the first read occurs directly after the initial write, the read-after-write lag and the measured staleness are almost equal, whereas a delay between the initial write and the first read that is greater than the data-centric staleness window would result in a zero-staleness measurement, because the first read would be successful. Since only few zero-staleness measurements occurred in our first experiment, the great number of large read-after-write lag values can only mean that there were many staleness measurements where the first read was executed within the first second after the initial write (resulting in an unsuccessful read), so that the reader had to push back the corresponding key to ZooKeeper, but then did not receive it again for a substantial amount of time (tens of seconds in some cases).

Our logs revealed that the YCSB++ reader actually achieved more than 320 reads per second, irrespective of the configured fraction, but also that it read the exact same keys dozens of times in succession instead of cycling through them as expected. Reviewing the YCSB++ code, we found out that the reader does not actually dequeue items from ZooKeeper, but instead uses the first item from the list that is returned by calling the `getChildren` method; according to our experiments (and the ZooKeeper documentation for various releases), though, this procedure does not guarantee the keys to be returned in any specific order which explains why the reader repeatedly dequeues the exact same keys. We verified through further experiments that the usefulness deteriorates even further when the fraction is increased: with the fraction set to 50%, virtually all staleness values are either way too large or zero. We managed to achieve a reasonable staleness approximation with YCSB++ at 1 staleness measurement per second (see Figure 5). However, this setting is not reliable: in an experiment with an increased data-centric staleness of 5000 ms demonstrate, the measured values were wildly scattered again, peaking at over 180 s.

Summary Using SickStore, we were able to uncover substantial imprecision in an experimental setup that would otherwise have been hard to detect. In particular, we could establish that the YCSB++ staleness measurement corresponds to neither the actual data-centric staleness nor to the real distribution of the staleness that is perceived by the reader. Further, we found out that YCSB++ provides barely any control over the degree of workload disruption, even though this is advertised in the YCSB++ paper: the reader always runs

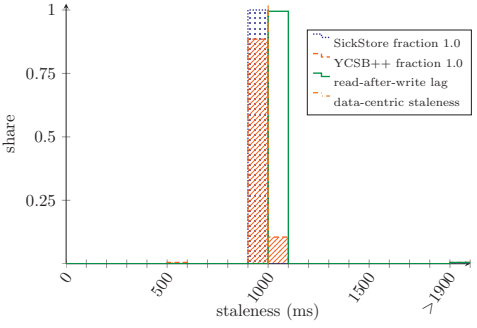


Figure 5: Target 1 ops/sec, fraction 100%: it is possible to achieve reasonable results.

unthrottled, irrespective of the configured fraction. The most severe problem for staleness measurement is that the YCSB++ reader mostly enqueues and dequeues the exact same keys in direct succession, so that some keys are not requested at all for tens of seconds which leads to significant distortions. Depending on the number of keys stored at ZooKeeper as well as the actually exhibited staleness, the resulting overhead may completely dominate the experimental results as it can be orders of magnitudes greater than the value that should actually be measured.

5 Conclusion and Open Challenges

The great variety of feature sets that are available on the datastore market also brings confusion with respect to what guarantees a datastore can actually provide, particularly with respect to performance characteristics that are hard to quantify objectively. To enable an evaluation of the different benchmarks that try to address this issue, we presented our prototypical database for validation, **SickStore**, that mimics the behaviour of a distributed a database system and simulates anomalies, but runs on a single server, so that all events can be associated to global timestamps without any clock drift. We described SickStore's architecture and its main features. In our experimental validation, we demonstrated how SickStore can be used to doublecheck staleness measurement results by comparing SickStore's internal logs to the YCSB++ measurements. Thus, we uncovered that YCSB++ is likely to produce misleading staleness values without the user noticing.
In order to enable validation of other measurements apart from staleness benchmarks, we are currently implementing freely configurable transactional anomalies as well as request latency and read/write throughput into SickStore. Further, we are building more sophisticated staleness generators that simulate different replication or sharding strategies and allow a more fine-grained tuning of staleness between the virtual SickStore server nodes.

References

[BT14] David Bermbach and Stefan Tai. Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies. In *Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2014. Best Paper Runner Up Award.

[BZS14] David Bermbach, Liang Zhao, and Sherif Sakr. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In *Performance Characterization and Benchmarking*, volume 8391 of *Lecture Notes in Computer Science*, pages 32–47. Springer International Publishing, 2014.

[CS+10] Brian F. Cooper, Adam Silberstein, et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154. ACM, 2010.

[FW+14] Steffen Friedrich, Wolfram Wingerath, et al. NoSQL OLTP Benchmarking: A Survey. In *Informatik 2014*, volume 232 of *Lecture Notes in Informatics (LNI)*, pages 693–704. Gesellschaft für Informatik (GI), 2014.

[PP+11] Swapnil Patil, Milo Polte, et al. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14. ACM, 2011.

Query Optimization

Plan Operator Specialization using Reflective Compiler Techniques

Carl-Philip Haensch, Thomas Kissinger, Dirk Habich, Wolfgang Lehner

Technische Universität Dresden
Database Technology Group
Dresden, Germany
{firstname.lastname}@tu-dresden.de

Abstract: Query-specific code generation has become a well-established approach to speed up query execution. However, this approach has two major drawbacks: (1) code generators are in general hard to write and maintain, (2) code generators lack the ability to deal with custom operators. To overcome these limitations, we suggest to return to the traditional execution approach with precompiled generic operators which are parametrized and composed to query plans at query compile time. Nevertheless, to optimize such plan operators and speed up their execution, we introduce a novel specialization approach using reflective compiler techniques. Employing code annotations and an additional compiler pass, we are able to track and replace low-level load instructions that refer to operator parameters which remain constant during execution time. By dissolving such up-to-now unknown constant variables, the compiler can further optimize the code and is able to determine query-specific optimized operators out of generic operator code. In our evaluation, we show that our approach speeds up the execution of the traditional generic operator approach in terms of execution time without facing the drawbacks of code generators.

1 Introduction

Due to the tremendous increase in the amount of data managed by today's database systems, query optimization is still one of the most challenging issues in database research. In disk-based database systems most of the time is spent for waiting on data from disk and thus, the effective CPU computation time is more or less a negligible factor. However, the ongoing trend towards in-memory databases shifted the bottleneck from disk access to the main memory access, resulting in a higher bandwidth and a lower latency when accessing data objects. Due to these decreased data access times, the effective CPU computation time now takes a significant share of the overall query execution time in such in-memory database systems. Therefore, besides determining an optimal query execution plan, the efficient execution of query plans including all operators attracted attention in the domain of in-memory database systems. The goal of this research is to execute query plans and its operators with less instructions to save as many CPU cycles as possible.

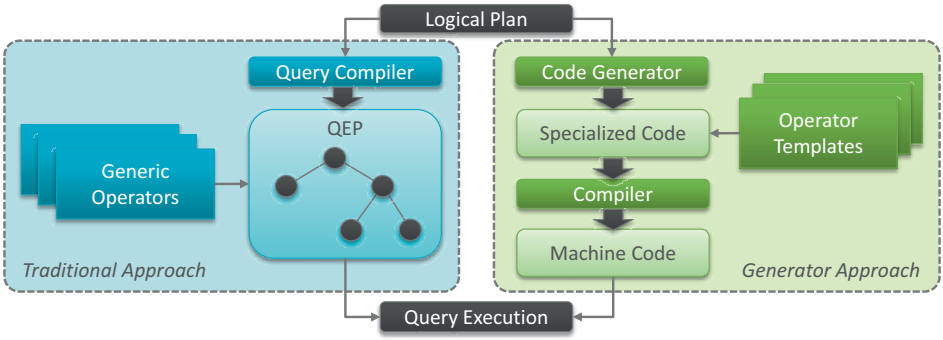


Figure 1: Comparison of Existing Query Compilation Approaches.

From a conceptual perspective, two different approaches for query execution have been developed as illustrated in Figure 1. On the left side of the figure, the traditional approach using a generic query operator code is shown. During each query compilation, a query execution plan (after query optimization) is composed using a set of precompiled generic operators and query-specific parameter settings for each plan operator. Afterwards, the plan is executed without any further optimizations. On the right side of Figure 1, the newly established generator approach is depicted, where a query–specific execution plan is generated using two main components: (a) a code generator and (b) operator templates. The advantages of such generator approaches are: (1) a highly specific query execution code is constructed and (2) this query execution code is further optimized using a compiler such as LLVM. Nevertheless, the significantly reduced execution time requires the costly development of hand-written code generators as well as operator code templates, whereas code generator and templates are hard to maintain and to extend. An additional disadvantage is the fact that the integration and optimization of custom code is challenging and not solved in any way.

To overcome the disadvantages of the generator approach without losing its performance benefits, we suggest in this paper to return to the traditional approach and extend it by a novel specialization concept at query compile-time using reflective compiler techniques. The general advantage compared to existing approaches is our novel architecture that combines the generic way of developing operators with the option to specialize operators fully automatically for a given execution plan which enables an easy development, maintenance and extension of operators. Moreover, this architecture even allows us to easily specialize a custom operator code which is not known at database compile time. To perform this specialization, we introduce code annotations to enable operator developers to mark certain parameters in their code as remaining constant after query compilation. Using these code annotations, the knowledge about the generic operator code and the actual parameters, we are able to further optimize the code. This happens with the help of a novel compiler pass that dissolves the constant behavior of parameters, so that we are able to generate query-specific optimized operators out of generic operator code. As we are going to show, using our constant annotations, we are able to specialize generic operators, which significantly speeds up query execution.

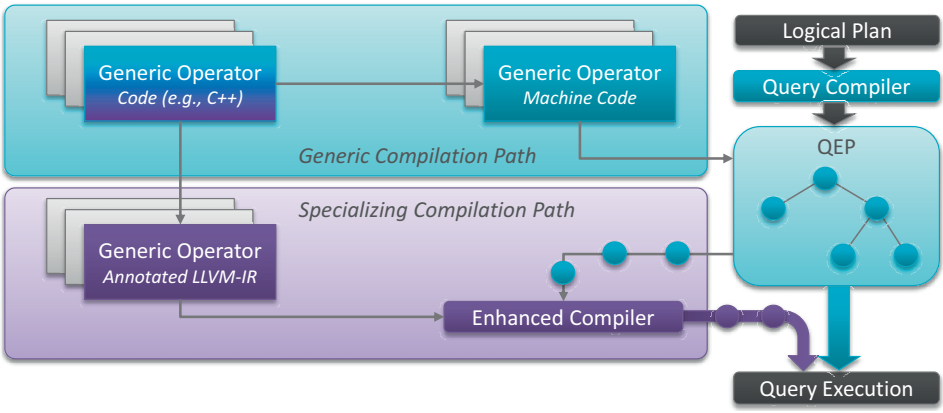


Figure 2: Hybrid Architecture Consisting of a Generic and Specializing Query Compilation Path.

The remainder of the paper is organized as follows: In Section 2, we are going to describe our hybrid architecture consisting of a generic and specializing query compilation path. Based on this overview, we introduce our plan operator specialization in Section 3. In this description, we use a minimal example to illustrate our approach, followed by a database operator example. At the end, we present our new compiler pass for plan operator specialization. Our evaluation in Section 4 is based on micro benchmarks as well as on the star schema benchmark to show the benefits. Before we conclude our paper in Section 6, we review related work in Section 5.

2 Architecture

In this section, we introduce our novel hybrid architecture that employs two different query compilation paths as depicted in Figure 2. The first path is the *Generic Compilation Path*, which takes the traditional way of composing a query execution plan (QEP) out of parametrized instances of generic plan operators. The additional second path is the *Specializing Compilation Path*, which is able to replace generic operator instances of the QEP by specialized query-specific ones. While current architectures either employ only one of those paths – facing their respective benefits and drawbacks – or both paths in a separated fashion, our novel architecture orchestrates both paths into a holistic query compilation framework that is able to choose between both query compilation approaches at the operator instance level. In the following we will detail on both compilation paths.

Generic Compilation Path This compilation path realizes the traditional way of QEP generation, where plan operators are implemented in a highly generic fashion to allow them to deal with all kinds of possible scenarios. Such generic operators are compiled to

executable machine code at database compile time and are instantiated and parametrized at query compile time to fit the actual requirements. Afterwards, all operator instances are composed into an executable QEP. Nowadays, when in-memory databases are becoming the standard, this traditional generic approach faces disadvantages in terms of query execution time because generically implemented operators exhibit a significant instruction overhead. However, common scenarios exist where the generic compilation path still outperforms the specializing compilation path due to the additional costs for invoking a compiler during code generation, e.g., for short-running ad-hoc queries, simple operators, or custom operators which can not be specialized by existing approaches.

Specializing Compilation Path This compilation path is able to transform parametrized instances of a generic operator, which were generated by the generic compilation path, into a specialized query-specific operator. Compared to existing approaches, which require the implementation and maintenance of a separate code generator, our novel approach tightly integrates with the generic compilation path. To do so, the first step is to enable plan operator developers to add annotations to the generic operator code that mark certain parameters as remaining constant after query compilation. As a second requirement, we need a representation of the generic operator code at database runtime that is as low-level as possible, but is still high-level enough to be efficiently optimizable. Hence, we decided to choose the LLVM Intermediate Representation (LLVM-IR) which is an intermediate result of the database compilation process and is used by the LLVM compiler infrastructure to apply several optimization passes. Thus, besides shipping the binary code of the generic operators with the DBMS, our framework additionally requires the inclusion of the LLVM-IR operator code in combination with the aforementioned annotations to allow the database system to reflect on its own operator code. To fully automatically compile specialized query-specific operator code, we take the LLVM-IR of a generic operator, corresponding annotations, as well as the actual parameters of a generic operator instance and feed them into our enhanced LLVM compiler version, which adds an additional optimization pass (cf. Section 3.3). Finally, we replace the generic operator instance by the specialized code if this is beneficial for the overall query execution time. This operator code specialization either happens synchronously at query compile time or asynchronously in parallel to the query execution, where generic operators are hot-swapped as soon as the specialized code is available.

3 Plan Operator Specialization

In this section, we will describe the process of plan operator specialization in three steps. First, we will examine a minimal example to develop a concept of algorithmically specializing generic operator code. Second, we will show with an example that the cognition obtained can also be applied to arbitrarily complex operators. Third, we describe how the specialization is actually implemented using the LLVM compiler framework.

3.1 Minimal Example

Generic operators in general consist of two parts: (1) A generic execution method and (2) parameters. While the generic execution method is already available during database compilation, the parameters are bound at query compile time. Figure 3 shows an operator after query compile time: There is an instance of a generic operator on the left side that already has its parameter bound. The generic execution method called `evaluate` (bottom of the figure) is written agnostically to the actual value of the parameter `x`. When considering the now known value of `x`, an other, smaller and therefore faster code would have the same semantics.

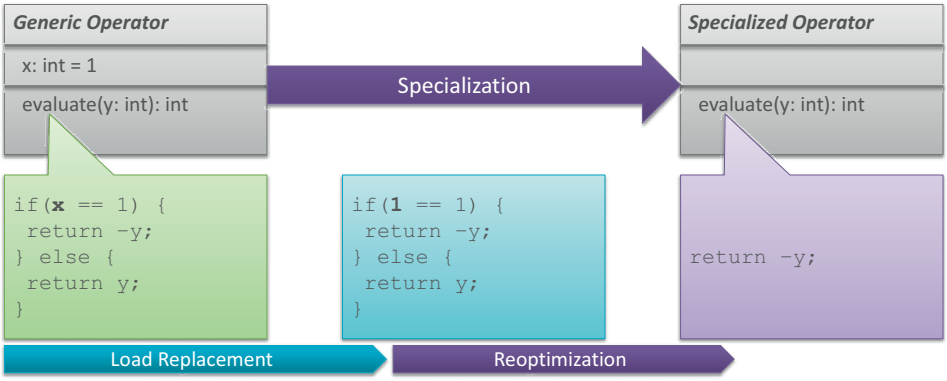


Figure 3: Minimal Specialization Example using A Simple Generic Operator.

High level programming language source code is rather unsuitable for our considerations. Such languages make use of a high variety of concepts. For instance, global variables, array elements, elements of a struct and attributes of an object all describe different concepts, but all describe a piece of memory that can be accessed through load instructions. For this reason we translated the example in Figure 3 into a pseudo assembly language as shown in Figure 4. This pseudo assembly only knows three kinds of instructions: arithmetics, jumps and load/store commands. The `if` condition is translated into a load instruction, a comparison and a conditional jump. The expression `return y` is translated into a load instruction and a jump and `return -y` is translated into a load instruction, an arithmetic instruction and a jump.

The left box of Figure 4 already shows an optimized assembly as it is spilled from a compiler. The problem with the generic operator is that it cannot be further optimized because the load instructions introduce data dependencies that cannot be resolved at database compile time. Since the parameter is known at query compile time, we propose to run the code optimizer of our compiler again over the code after the values of the parameters are known. To overcome optimization-blocking load instructions, we have to replace them by actual values.

After replacing `load x, %b` by `mov 1, %b`, traditional compiler passes are able to

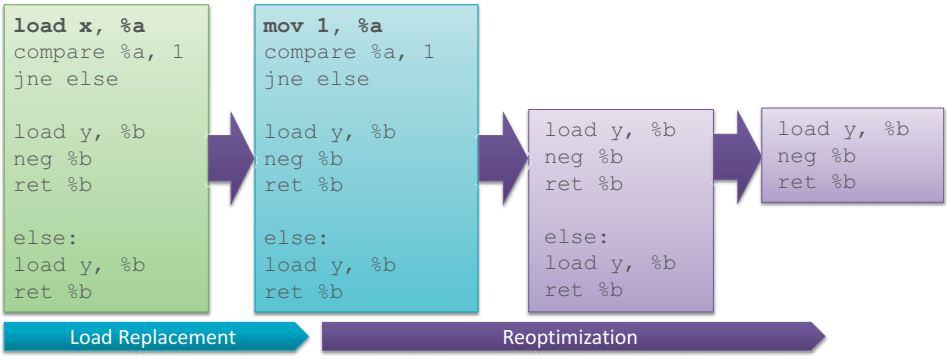


Figure 4: Minimal Example in Pseudo Assembly.

optimize our example even further: The *instruction combining* pass of our stock compiler is able to combine the `mov` and `compare` instruction and directly propagate *equal* instead of computing a comparison. The conditional jump operation `jne` can therefore be removed because the condition is never fulfilled. After removing the jump instruction, the label `else` is never reached and can therefore be removed, too.

In summary, we are able to specialize an operator completely just by using simple code transformation rules. We discovered that replacing load instructions of operator parameters by the values of the parameters offers a method of automatically creating specialized code without the need for writing a custom code generator.

3.2 Database Operator Example

The operator examined in section 3.1 is rather simple. Database operators are much more complex. They contain loops, nested branches and virtual function calls where template based code generators are likely to fail. This is not the case for our approach. We will show that in general lots of code primitives occuring in database operators are load instructions followed by some instructions that depend on the result of the load instruction. To make this tangible, we provide an example in Figure 5, which shows a generic group operator.

For our example we choose the following parameters: `group_attributes` has size 2. The first element has a `column` of `2` and the second element has a `column` of `3`. With the size parameter known, we can construct a specialized version of the loop such as in Figure 6. The method we used is called *loop unrolling*. It is a normal compiler pass that is implemented in nearly all modern optimizing compilers. The only reason why the loop cannot be unrolled was that `group_attributes.size()` was unknown. However, this value is constant and known for this operator instance after the query was compiled. After replacing the load instruction from the `size` field of the vector, the loop has a constant iteration count. The optimizer can now remove all control instructions and insert

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          for(int i = 0; i < group_attrs.size(); i++) {
4                  key.append(tuple.extract(
5                          group_attrs[i].column));
6          }
7          outputIndex.insert(key, tuple);
8  }
```

Figure 5: GROUP BY Operator Example Code.

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          key.append(tuple.extract(
4                  group_attrs[0].column));
5          key.append(tuple.extract(
6                  group_attrs[1].column));
7          outputIndex.insert(key, tuple);
8  }
```

Figure 6: GROUP BY Operator Example Code with Unrolled Loop.

the loop body n times. Figure 7 shows instruction-wise how big our savings are.

An interesting situation comes up when accessing arrays inside of loops. The upper example accesses `group_attributess` elements with the unknown index `i`. After unrolling the loop, this index is known and therefore a concrete memory address can be derived. Figure 8 shows the code of the group operator after specialization. Since this operator code (group by) is executed inside of a loop, even small savings in the instruction count yield remarkable speedups in the execution time. `tuple.extract` is a rather simple function. Inlining it would open up even more optimization potential. The following code patterns occur in real database operator code:

Loops with a Constant Number of Iterations Iterations over tuple elements are mostly operation specific and can therefore be specialized. Applying the specialization removes unneeded branching and unnecessary compare instructions. Also, unrolling loops opens new possibilities to access different data in each loop iteration, e.g., type information for each tuple element.

Constant Values (SQL Constants) Queries such as `SELECT x+1 FROM t` contain constants such as `1` which are better to be inlined than fetched from somewhere. Generic operators implement them as load instructions from some constant storage. Inlining these

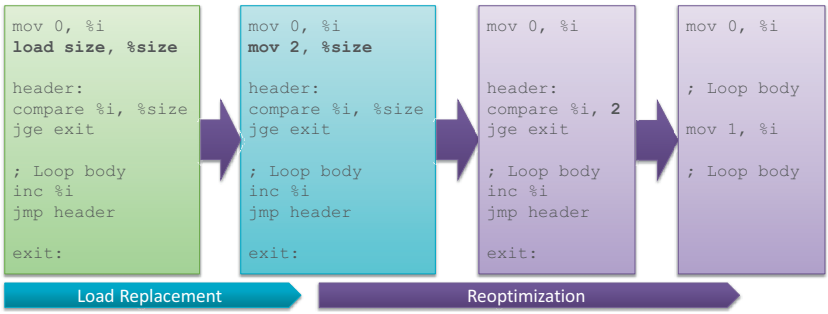


Figure 7: Loop can be Unrolled Because of Resolved Load Instruction.

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          key.append(tuple.extract(2));
4          key.append(tuple.extract(3));
5          outputIndex.insert(key, tuple);
6  }
```

Figure 8: GROUP BY Operator Example Code with Unrolled Loop and Inserted Parameters.

constants into the machine text also prepares other optimizations such as instruction combining. By using our approach, a SQL expression such as `2+2*2` could be computed during query compile time.

Virtual Function Calls Operators that are built calling virtual functions recursively. With an inlining pass, we can remove call overhead and also have the ability to perform interprocedural optimizations. One example is `tuple.extract` in Figure 8 where inlining the tuple extraction algorithm would open up additional optimization potential. Combined with inlining `key.append` even more aggressive optimizations would be possible. Fetching a function pointer from a virtual function table is a load instruction. We detect this pattern and inline the function call. For ahead-of-time compilers, virtual function inlining is not easy. In the worst case, all code has to be scanned, which is not always possible. Given a specific runtime object, virtual function inlining hence is trivial.

Branches Branching occurs when compiling control flow statements such as `if` and `while`. For branches that depend on operator parameters rather than input, we are able to remove branching and therefore remove unnecessary comparisons, jumps and dead code.

And a lot more Patterns ... Our approach does not handle constants, loop iteration counts, data type selectors or tuple offsets as corner cases. We rather see that replacing load instructions by constants, optimization passes already available in stock LLVM are triggered and specialize the code independent from its structure. In summary, we are able to deal with arbitrary complex generic operator code. All code is compiled to machine code that contains load instructions, jumps and arithmetic instructions.

3.3 Specializing Compilation Path

In this section, we will present our novel compiler pass. In the first subsection, we will give an overview on how this pass interacts with other passes inside the compilation process. In the second section, we will describe the algorithm behind our pass.

3.3.1 The Compilation Process

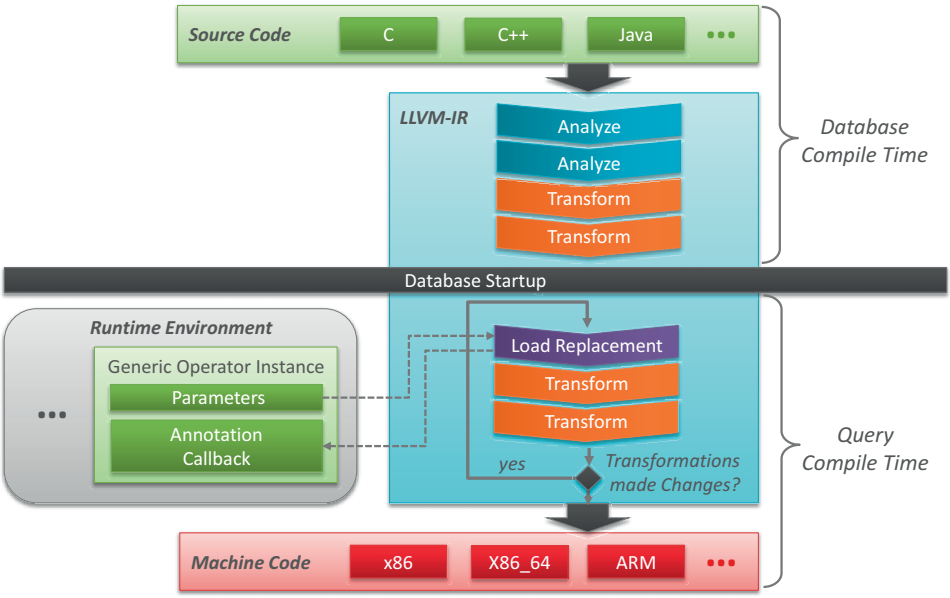


Figure 9: LLVM Compilation Process with Operator Specialization.

The traditional database compilation process is restricted to database compile time. Our approach also involves query compile time. The compiled code of the generic operators in their intermediate representation is kept for further specializations. During each query compilation, this code can be further specialized due to bound operator parameters.

We specialize operators by involving their already bound parameters. This happens by

loading the already optimized generic code (created at database compile time) and running compiler passes over it (Figure 9). Except from our *load replacement* pass we did not add any functionality to LLVM[1]. All optimizations are already known in compiler construction science. The *load replacement* pass opens up new optimization possibilities when traditional compiler passes can not further optimize the code. Therefore we try to run all beneficial traditional compiler passes over the code until they resign. Then we run our novel pass such that traditional passes can continue optimizing the code.

The first pass running over the code is the *load replacement*. We start with this pass because it removes load instructions that blocked further optimizations during database compile time and replaces them with compiler-friendly constants. After that, *instruction combining* can do traditional constant propagation in order to partially evaluate branch conditions and arithmetics. After that, *control flow graph simplification* can remove dead code and reorder the basic blocks to an optimal order.

For more complex operators a different pass order is applied. Operators containing loops have to be handled separately. We also begin with *load replacement*. After *instruction combining* some loop counts are known, so a *loop unrolling* pass can be run. After that, *load replacement* has to be run again because array access inside the loop makes use of the control variable of the loop. A load instruction can only be replaced by a constant when the memory address is constant which does not hold for addresses that are calculated from the control variable. After unrolling, new load instructions can be resolved. After loops are unrolled and their inner load instructions are resolved, *instruction combining* can be run again and *control flow graph simplification* cleans up the specialized code. The overall result after the specialization process is much smaller code. With less instructions, the code fits better into the instruction cache and can be executed faster. To save time, we do not run loop passes on rather simple operators such as arithmetics.

We also run the *virtual function call inlining* pass for functions that call other functions. This pass is placed before the last *instruction combining* pass to reveal further optimizable code patterns to the following passes. It enables us to inline deep expressions like `(2*i) + 1` directly into one function.

3.3.2 Load Replacement Pass

This pass is the key part of our approach. It pushes operator parameters into the generic operator code. In object oriented programming, a method is a function that operates in the context of an object. Most programming languages implement this by passing a parameter `this` to the function. The operators themselves are classes where some of the attributes are operator parameters. To achieve operator specialization, the *load replacement* pass has to be aware of the value of the `this` parameter of the operator method. Load instructions relative to `this` load attributes of the object which can by chance be constant operator parameters. To assure that an attribute of the object is in fact constant, we have to annotate the memory somehow.

[1] *virtual function inlining* is also written by us because we want to inline already optimized operators into their parent operators.

The pass itself iterates over all instructions of a method and looks for load instructions. For each load instruction it tries to resolve the memory address to load from. This is done by reverse traversion of the code. The memory address is calculated from some base value (e.g., `this`) and offsets. If some of the offsets are not constant but depend on a loop control variable or a load instruction, we can not calculate the address. In case we found a load instruction that loads from $mem[\texttt{this} + offset]$, $mem[mem[\texttt{this} + offset] + offset]$ or any deeper indirection, calculating the address is possible. These addresses are constructed from LLVM-IR's instructions `load`, `bitcast` and `getelementptr`, as well as `%this` at the end of the chain. By handling them in a recursive function, we can resolve all memory addresses to attributes in the code. Once the address is calculated from all offsets, we consult our annotation system whether there is a constant at this memory location or not. If this is the case, the load instruction can be executed at compile time by loading the value from runtime memory.

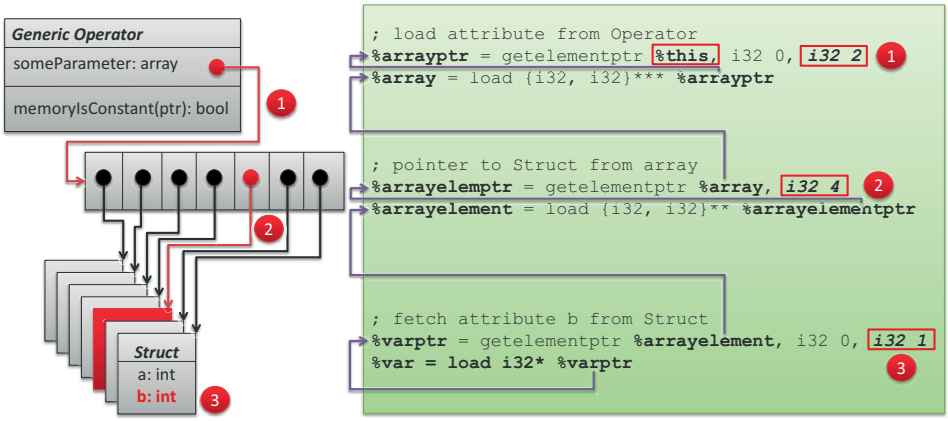


Figure 10: Loading Operator Parameters in LLVM-IR.

Figure 10 illustrates what operator parameters could look like. The operator parameter `%var`, in C `someParameter[4]->b`, is hidden behind two indirect loads. When the value of `%this` is known (and the offset is constant), the address `%varptr` can be computed. Computing the address is implemented by resolving `%varptr` recursively. `load` instructions in indirect memory addressing such as `%arrayelement` have to be executed during pass runtime. This is only allowed when the annotation system marks the loaded memory constant. A `getelementptr` instruction simply adds an offset to a pointer. The then computed address is passed to our annotation. If the annotation assures constantness of the memory at this address, an integer constant can be fetched from the address `%varptr`. This constant is packed into a LLVM value and all occurances of `%var` are replaced by that constant. Subsequent passes will remove the instructions that compute `%varptr` too when they are not referenced any more.

The annotation is managed by the operator (`this`) and is realized through a method called `bool memoryIsConstant(void *mem)` which returns *true* for a given memory

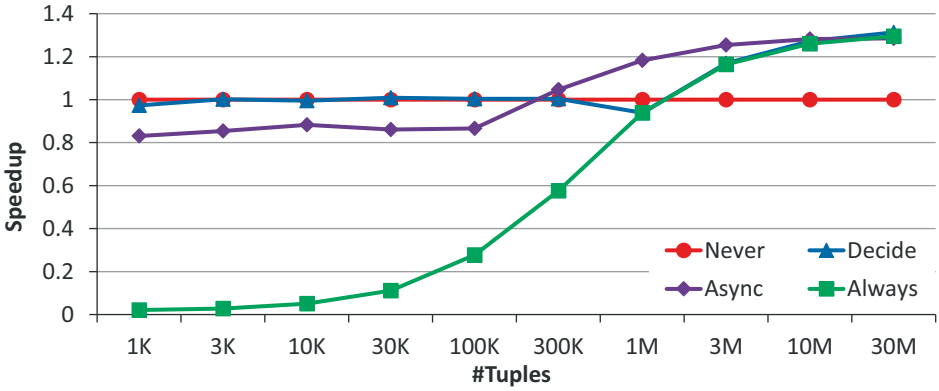


Figure 11: Execution Time Speedup of Ad-hoc Queries for Micro Benchmark with Different Data Sizes and Different Specialization Strategies.

location when there is a constant at this address. This way, the memory is mapped into regions that can change and regions that remain constant. The granularity of this mapping is byte-wise. The annotation system does not care about variables, attributes or data types. It also is not compiled into the LLVM code. Instead, it just manages a mapping that tells at which memory address there is a constant. Determining which address is constant is delegated to the operator objects that know which of their attributes are constant. The address ranges of the attributes are compared to the address queried by the annotation manager.

4 Evaluation

In column stores, operators work vector-wise[RBZ13]. This reduces complexity and makes it possible to generate all variations of operator parameters, such as input types, with templates. However, this is not possible for tuple based operators since the number of variations grows exponentially with the number of elements in a tuple.

We used two benchmarks to evaluate the performance of our specialized code. The first one is a micro benchmark that covers all combinations of the sum aggregation with single value addition. The second one is Star Schema Benchmark (SSB) which is widely used in data base research[OOCR09]. As a test setup, we used DexterDB, which was developed by the database research team of TU Dresden. It is an in-memory database, which has an indexed table-at-a-time processing model[KSHL13]. We implemented our approach into the already optimized operator system of DexterDB and measured what speedup we can achieve with our additional operator specialization.

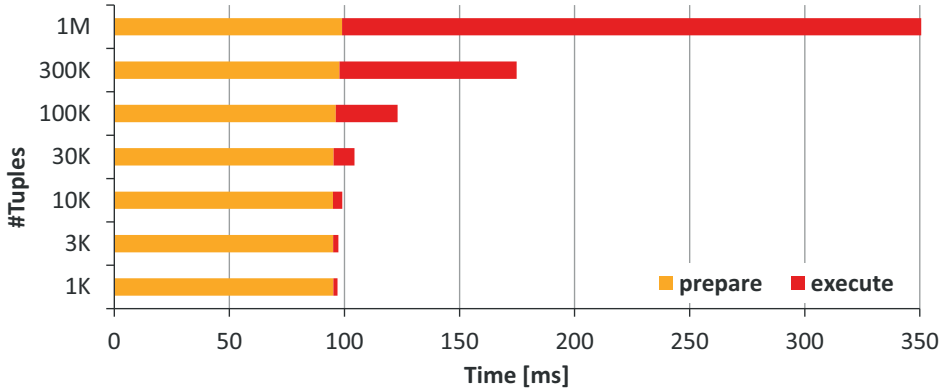


Figure 12: Execution and Prepare Time of Ad-hoc Statements for Micro Benchmark with Different Data Sizes.

4.1 Micro Benchmark

The micro benchmark consists of 5 queries which cover simple aggregation, aggregation combined with arithmetics (several bracketings), aggregation with a condition on each input tuple and a simple join with a table of 5 elements. All queries are executed together to get a median over some common queries. The benchmark varies over the size N of one input table.

All queries are executed 32 times with each of these four specialization strategies: **Never** strategy does never specialize and reflects the unoptimized result of the generic operators. **Always** reflects the result with specialized operators. **Decide** tries to find a balance between execution and preparation effort and only specializes when the input tables are big enough (threshold of N). **Async** runs the specialization in an extra thread and asynchronously swaps methods when the query runs a long time but cancels specialization if the query was faster.

The test machine is an Intel(R) Core(TM) i7-3960X CPU with 6 cores each 3.30 GHz with a total of 12 hyperthreads running linux. L1 instruction and data cache are 32 KiB each. L2 cache is 256 KiB and L3 is 15360 KiB.

One measurement consists of executing all 5 queries 32 times with table size N. On each execution two measures are taken: prepare and execute time. Prepare time is the time from parsing the statement, optimizing and specializing until it is ready for execution. During prepare time no data is touched. Prepare time is also the time it would take when a prepared statement is created. Execution time is the time it took to execute. Executing an ad-hoc query would take prepare+execute time while executing prepared statements would only take the execute time.

Preparing unspecialized queries took about 0.2-0.5 ms in total. However, preparing specialized queries however took about 95-99 ms for all five queries, which is quite costly

in comparison to unoptmized queries. Figure 12 shows that code generation is expensive, which especially concerns small queries. Therefore two additional strategies for handling small queries were introduced: **Decide** and **Async**. As one can see in Figure 11, these strategies give good results for both big and small queries. Figure 11 shows how they combine the advantages of both code paths: the generic execution path and the specialized operator path. While decide ammortizes relatively late, the asynchronous strategy already starts gaining advantage at about 100,000 items. The downside of the **Async** strategy is that each query takes some extra milliseconds for synchronization. **Async** has to stop the specialization thread which is not possible at any time.

4.2 Star Schema Benchmark

The Star Schema Benchmark[OOCR09] (SSB) is a benchmark derived from TPC-H. It aims at benchmarking database products that run classical data warehouse applications. The benchmark consists of 8 tables and 13 queries containing aggregations, joins and selections.

	Generic		Specialized	
Nr.	Prepare Time	Execution Time	Prepare Time	Execution Time
Q1.1	0.20 ms	8,312.01 ms	41.29 ms	4,979.34 ms
Q1.2	0.21 ms	8,734.92 ms	43.43 ms	5,319.47 ms
Q1.3	0.21 ms	8,692.01 ms	45.43 ms	5,357.07 ms
Q2.1	0.20 ms	10,900.00 ms	77.29 ms	9,117.51 ms
Q2.2	0.22 ms	7,758.99 ms	80.77 ms	6,098.14 ms
Q2.3	0.18 ms	5,508.16 ms	78.31 ms	3,907.43 ms
Q3.1	0.22 ms	11,163.30 ms	91.54 ms	8,766.37 ms
Q3.2	0.22 ms	7,285.59 ms	91.87 ms	5,690.01 ms
Q3.3	0.23 ms	5,574.93 ms	91.80 ms	4,131.47 ms
Q3.4	0.22 ms	5,805.91 ms	90.81 ms	4,121.73 ms
Q4.1	0.25 ms	10,382.60 ms	99.21 ms	8,203.17 ms
Q4.2	0.30 ms	9,797.42 ms	112.60 ms	7,742.18 ms
Q4.3	0.28 ms	6,896.63 ms	108.98 ms	5,174.66 ms

Figure 13: Results of the SSB-Benchmark.

The benchmark is executed with scale factor 20 which fits into 32 GiB RAM of the test machine. Figure 13 shows the results of our measurements. When executing queries with a large ammount of data, the average 70 ms of prepare time does not influence the overall result that much. Figure 14 especially shows that, in contrast to the micro benchmark, the difference in speedup between prepared statements and ad-hoc queries is not remarkable for big data queries like those from the SSB. This is also confirmed in Figure 15 where we ran SSB with different scale factors.

Figure 15 shows two SSB queries (Q1.1 and Q2.2) that we benchmarked with different scale factors both with and without indices. Figure 16 illustrates the speedup from Fig-

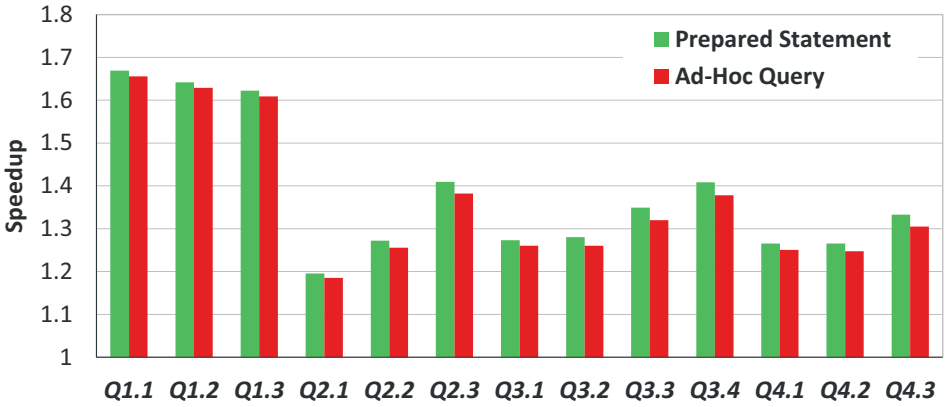
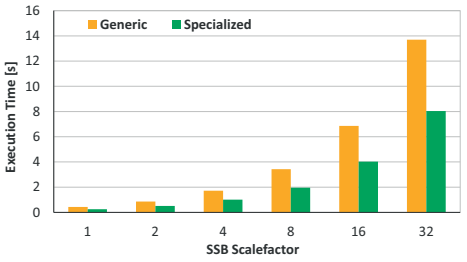
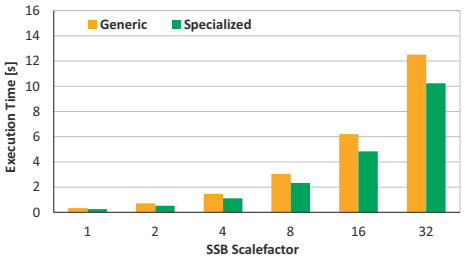


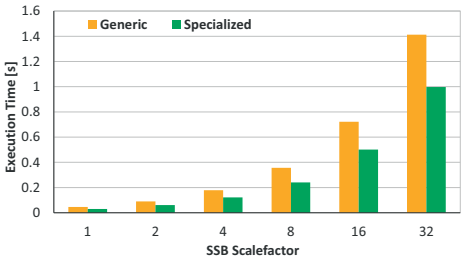
Figure 14: Speedup of Specialized SSB Queries.



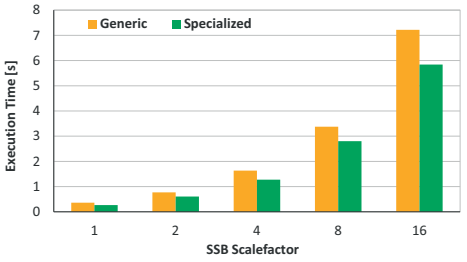
(a) SSB Query 1.1 w/o Indexing



(b) SSB Query 2.2 w/o Indexing



(c) SSB Query 1.1 w/ Indexing



(d) SSB Query 2.2 w/ Indexing

Figure 15: Execution Times of SSB Queries for Different Scale Factors and Storage Configurations.

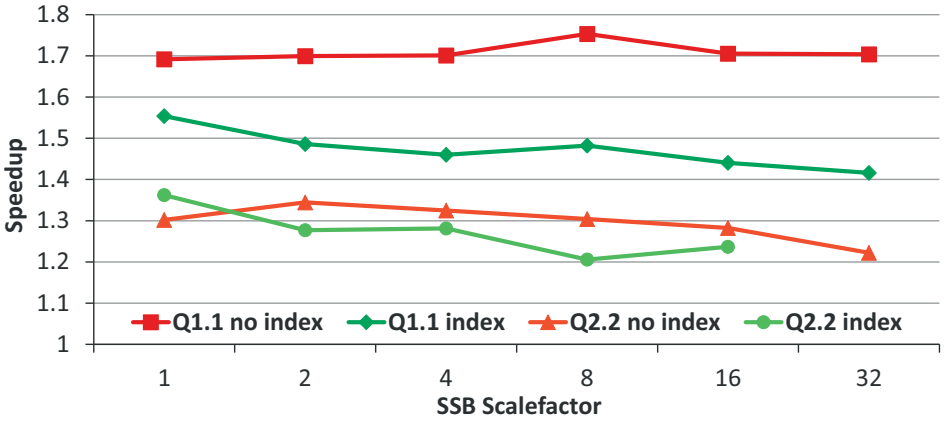


Figure 16: Speedup of Specialized SSB Queries 1.1 and 2.2 w/ and w/o indexing for Different Scale Factors.

ure 15's runs in one chart. While the run of Q1.1 without indexing has a rather constant speedup, speedup sinks in the other runs. Q1.1 does not have joins and therefore just iterates over a big ammount of data. We achieve the highest speedup with this setting. With enabled indexing, speedup is worse but still good. The declining speedup for enabled indexing and Q2.2, which consists of some joins, can be explained with smarter algorithms that avoid raw computations and therefore do not offer large optimization potential.

Our approach achieves a speedup between 1.2 and 1.6 in DexterDB for SSB which is a favorable result. Especially Q1.1, Q1.2 and Q1.3 perform great which can be attributed to their heavy algebraic load. The other queries rely more on multi way joins. With an additional recursion unroller, which we did not write, we could further speed up joins.

5 Related Work

There are various approaches to achieve low-level optimized operators. One approach is to transform and specialize existing code. An other approach is to generate already specialized code using code generators.

5.1 Code Specialization Approaches

Our approach is one instance of partial evaluation[2]. Our compiler pass partially executes generic operator code and generates the leftover specialized code. While we work with annotations for marking constants, there are also approaches to have partial evaluation as a language feature. The Java Programming language[GJSB05] for instance supports the `final` flag for variables which has exactly the semantics of our annotation. A Java Runtime Engine could specialize methods of an instanciated object in a background thread and inline the actual values of the final fields into the method code. The problem with this feature is that it is hard to decide whether a specialization is beneficial. Speculatively specializing code often yields performance penalties. A runtime engine can not determine how often code is executed in general. However, in our domain we have this information and use it to decide whether to specialize or not.

Partial evaluation is something Veldhuizen[Vel99] already described in 1999. Veldhuizen's approach is based on C++ templates which are restricted to integers at compile time. What he predicted is that it will be possible to do partial evaluation based on bind time data. Our approach is even better, since it can do partial evaluation during program runtime.

One approach did come close to ours. Skye Wanderman-Milne and Nong Li[WML14] proposed a complete architecture for code snippet composition based on LLVM called Cloud Impala. Parts of their code is generated by a hand-written code generator while other parts are C/C++ or Python code compiled to LLVM-IR and specialized for specific operator parameters. They make very specialized differenciations between conditionals, constant loads, runtime type information and try to manually specialize prepared code based on variable names. However, we discovered that any of these patterns contains a unresolved load instruction. Based on memory annotation, we can tell which load instructions can be resolved. Therefore we don't need extra variable loaders and can instead read out our runtime memory which makes our solution much more elegant. Also, their approach requires a completely new infrastructure for generic operators, which is not optimal for unspecialized code. Our approach allows further specialization of already optimized legacy generic operators.

5.2 Code Generators

Zhang et al.[ZSD12] introduced the term *micro-specialization* in DBMSs. While most optimization approaches focus on high level components, they focussed on parameterized operators in the classical operator approach. They focussed on variables - typically schema metadata or query-specific constants - that are invariant to the query evaluation loop. Micro optimizations remove unnecessary operations in frequently taken execution paths. The paper states that micro specialization is not possible using traditional compiler techniques

[2]Partial Evaluation means to not evaluate a statement or a piece of code completely. Instead, parts of the code are executed and the code left to execute is returned. For instance partially executing an `if` statement would mean to evaluate the branch condition and return only the code of the chosen path.

since the invariants are only known at runtime. Until this point the paper does not distinguish from this work. The difference is that Zhang et al. work with hand-written code templates while we derive our information from the original source code of the DBMS using compiler techniques. For us, the generic operator code is the template. Also, they are still restricted operator boundaries while we can inline code. They used code templates for some hot code operators which have to be specified by the developer. The developer has to define code snippets that are then assembled to so called bee routines. The gcc compiler is used to create the precompiled bee routines. Constants are inserted with preprocessor macros. For some values, so called magic numbers (1000, 1001, 1002) are used to be replaced by constants after loading the machine text into memory. The bee routines are loaded into memory and are then further specialized with constant values. Integrated into PostgreSQL they got a performance gain of about 12%, in rare cases even 33%. One thing that their paper does better than this paper is the preparation time. Loading a piece of machine code into RAM and replacing some constants is much faster than generating specialized code from the beginning.

Rao et al.[RPML06] implemented a query compiler for the Java Virtual Machine (JVM). They used it in their in-memory database called JAMDB. Besides interpreting a Query Execution Plan they generate java bytecode. Their implementation is said to run faster than the non-specialized C/C++ code from DB2[3]. They implemented both an interpreted engine and a JVM based compiler. This led to double expense of maintaining code and optimizing parts of the code. They admitted that this is harder to code and harder to maintain than simple interpreters.

Neumann[Neu11] proposes a completely new architecture of query processing that tries to keep all values of a processed tuple in CPU registers. This way, highly effective pipelining of multiple operations can be achieved. They used the LLVM framework to merge operations into one machine code function. They used a code generator to build specialized query plans on demand. Their generated code is said to be near optimal assembly code. Their code generator for full SQL-92 operators is about 11,000 lines of code which is not a lot in their perception. We used about 1,000 lines of code with our generalized approach (plus the code of the interpreted operators) which is even better.

The big advantage of our approach is flexibility. While other approaches have to maintain code generators which are harder to write, we can stick to generic operator code. Extending the execution engine with new operators consisting of custom code is possible and easy with our approach since every operator behaves like custom code to us. By using compiler techniques we are able to create optimized code with a minimal invasive programming model. The downside is a higher preparation time. Transforming generic code to specialized code is slower than generating only the needed parts of the optimal query execution code.

[3] It ran with a large bufferpool to have in-memory characteristics.

6 Conclusion and Future Work

We developed a hybrid architecture that can be put on top of traditional generic operator systems. This architecture enables us to specialize operators to their parameters when the benefit exceeds the effort of the specialization. This way we can speed up long running database queries.

Our framework takes the intermediate code of generic operators after database compilation and specializes them to a specific instance of that operator. This is done by resolving load instructions. Since the pointer to the instance of an operator is known to us, we know the memory locations of immutable operator parameters and can therefore resolve load instructions that load from these memory locations. This way, we are able to push additional information about the operator into the generic operator code, which enables further optimizations on the code to traditional compiler passes.

Adopted to a database which runs operator code millions of times, we measured speedups ranging from 1.2 to 2.0 in query processing. With these speedups, we revived the classical operator model and modernized it to meet the needs of modern in-memory databases. An implementation of about 1,000 lines of code is able to specialize arbitrarily complex operators independent of their semantics. This is useful especially when dealing with custom operators.

Our approach solves the same problem as hand-written code generators. However, we provide a much more elegant way to achieve a specialization. DBMS implementors do not have to write code generators from scratch to get speedup. They can just put their generic operator code into our algorithm and our enhanced compiler will specialize it to a certain environment, in this case to bound operator parameters.

This approach is not restricted to DBMSs. Load replacement enables to specialize generic code to any situation that can be encoded in memory. The reason why it is not widely adopted by scripting language interpreters, regular expression interpreters, image filters and other possible applications is the high effort for compilation. A problem with an automation of the specialization, for instance as a feature of a language's runtime, is to decide how often some specializable code is executed, which often depends on the size of the input. Compilation is expensive and speculative specialization can slow down execution when the specialized code is only executed a few times.

References

[Bar09] Edward Barrett. 3c - A JIT Compiler with LLVM. Technical report, Bournemouth University, 2009.

[Dan99] Olivier Danvy, editor. *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*. University of Aarhus, 1999.

[Fan10] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[KSHL13] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. QPPT: Query Processing on Prefix Trees. In *CIDR*. www.cidrdb.org, 2013.

[LA03] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[NW08] George C. Necula and Philip Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008.

[OOCR09] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[RPML06] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, page 23, 2006.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[Str07] Bjarne Stroustrup. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.

[UL08] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of java's final fields. In Necula and Wadler [NW08], pages 183–195.

[Vel99] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In Danvy [Dan99], pages 13–18.

[WML14] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, pages 31–37, 2014.

[ZNMZ13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal Verification of SSA Optimizations for LLVM, June 2013.

[ZSD12] Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Micro-Specialization in DBMSes. In *ICDE*, pages 690–701, 2012.

Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper
Technische Universität München
Munich, Germany
neumann@in.tum.de, kemper@in.tum.de

Abstract: SQL-99 allows for nested subqueries at nearly all places within a query. From a user's point of view, nested queries can greatly simplify the formulation of complex queries. However, nested queries that are correlated with the outer queries frequently lead to dependent joins with nested loops evaluations and thus poor performance.

Existing systems therefore use a number of heuristics to *unnest* these queries, i.e., de-correlate them. These unnesting techniques can greatly speed up query processing, but are usually limited to certain classes of queries. To the best of our knowledge no existing system can de-correlate queries in the general case. We present a generic approach for unnesting arbitrary queries. As a result, the de-correlated queries allow for much simpler and much more efficient query evaluation.

1 Introduction

Subqueries are frequently used in SQL queries to simplify query formulation. Consider for our running examples the following schema:

- students: {[id, name, major, year, ...]}
- exams: {[sid, course, curriculum, date, ...]}

Then the following is a nested query to find for each student the best exams (according to the German grading system where lower numbers are better):

```
Q1: select s.name,e.course
    from   students s,exams e
    where  s.id=e.sid and
           e.grade=(select min(e2.grade)
                    from exams e2
                    where s.id=e2.sid)
```

Conceptually, for each student, exam pair (s, e) it determines, in the subquery, whether or not this particular exam e has the best grade of all exams of this particular student s.

From a performance point of view the query is not so nice, as the subquery has to be re-evaluated for every student, exam pair. From a technical perspective the query contains a

dependent join, i.e., a nested loop join where the evaluation of the right hand side depends on the current value of the left-hand side. These joins are highly inefficient, and lead to (at least) quadratic execution time.

Database management systems (DBMSs) therefore internally rewrite the query to eliminate the correlation. A SQL representation of this rewrite would look like this:

```
Q1': select s.name,e.course
     from   students s,exams e,
            (select e2.sid as id, min(e2.grade) as best
             from exams e2
             group by e2.sid) m
     where  s.id=e.sid and m.id=s.id and
            e.grade=m.best
```

Here, the evaluation of the subquery no longer depends on the values of `s`, and thus regular joins can be used. This kind of unnesting is very important for good query performance, but existing techniques cannot handle arbitrary queries. For example the subsequent SQL query is very hard to de-correlate. It determines the exams that a CS or Games Engineering student should repeat in the future because he or she underachieved in comparison to the average grade of exams taken by him/her or taken by elder peers:

```
Q2:
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
  (s.major = 'CS' or s.major = 'Games Eng') and
  e.grade>=(select avg(e2.grade)+1 --one grade worse
            from exams e2          --than the average grade
            where s.id=e2.sid or   --of exams taken by
                  (e2.curriculum=s.major and --him/her or taken
                  s.year>e2.date))           --by elder peers
```

To the best of our knowledge, no existing system can unnest such a query. And indeed, unnesting this query is hard: Standard unnesting techniques rely upon the fact that attributes available within the query can be used to substitute the free variables determined by the outer query. This is not the case here, `s.year` for example cannot be substituted.

So clearly this kind of complicated correlated query will be more expensive to evaluate than a more simple subquery. However, as we will show, it is indeed possible to unnest even this query. We will have to spend extra effort to derive the value of `s.year` and `s.major`, but we can do so without a dependent join. And the extra effort we will have to spend is bound by the cost of the dependent join. Most queries will be dramatically more efficient in the decorrelated form, in the worst case we will have the some join effort. That is, our unnesting approach will definitely not incur higher costs than the straightforward nested loops evaluation – and in the majority of cases improve the performance dramatically, often by several orders of magnitude. Furthermore, even the worst case is

most likely a win, as eliminating dependent joins allows for more efficient join implementations. Our contribution can thus be seen as a universally applicable technique for unnesting any kind of nested subquery – in contrast to the special case treatments published and implemented so far. The universal unnesting technique has been fully implemented in our main-memory database system HyPer [KN11] and can be experienced via our web interface `hyper-db.de` that visualizes the resulting query plans.

And the typical performance gains of query unnesting are immense: Depending on the query, it replaces an $O(n^2)$ algorithm (nested loop join) with an $O(n)$ algorithm (hash join, joining keys). Furthermore the dependent side is executed for every outer tuple in the nested case, but only once in the unnested case. On large data sets it is easy to get a factor 10 or even 100 performance improvement by unnesting, which makes unnesting an essential technique for query compilation. There are a few cases where nested evaluation is actually beneficial, in particular if the outer side is very small and the inner side can be evaluated using an index lookup, but that should be triggered by a conscious decision of the query optimizer, not by the way the query is formulated. By default, queries should be unnested completely.

The rest of this paper is structured as follows: We first define the notation used in this paper in Section 2. Then, in Section 3 the algebraic unnesting transformations are specified. Section 4 covers further optimisation rules that are applicable in special cases (e.g., when functional dependencies can be inferred). Section 5 is devoted to a "cursory" performance evaluation that analyses some other well-known DBMSs to our HyPer system which incorporates the unnesting described. Finally, we survey the related work and conclude the paper.

2 Preliminaries

Before looking at the unnesting techniques, we briefly repeat some definitions for relational algebra, as the notation is not standardized beyond the basic operators.

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

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

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

$$T_1 \ltimes\!\!\!\bowtie_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

Here, the right hand side is evaluated for every tuple of the left hand side. We denote the attributes produced by an expression T by $\mathcal{A}(T)$, and free variables occurring in an expression T by $\mathcal{F}(T)$. To evaluate the dependent join, $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ must hold, i.e., the attributes required by T_2 must be produced by T_1.

Note that in this paper we sometimes explicitly mention *natural join* in the join predicate to simplify the notation. We assume that all relations occuring in a query will have unique attribute names, even if they reference the same physical table, thus $A \bowtie B \equiv A \times B$. However, if we explicitly reference the same relation name twice, and call for the natural join, then the attribute columns with the same name are compared, and the duplicate columns are projected out. Consider, for example:

$$(A \bowtie C) \bowtie_{p \wedge \text{natural join } C} (B \bowtie C)$$

Here, the top-most join checks both the predicate p and compares the columns of C that come from both sides (and eliminates one of the two copies of C's columns).

For semi joins ($\ltimes$), anti joins ($\triangleright$), and outer joins ($⟕$, $⟖$) we define the dependent variants accordingly ($⋉$, $\blacktriangleright$, $⟕$, $⟖$), again the right-hand side is evaluated for every tuple of the left-hand side.

Besides the join operators, we have the *group by* operator as additional important operator

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

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

We can evaluate functions (and thus construct new attributes) by evaluating the *map* operator

$$\chi_{a:f}(e) \quad := \quad \{x \circ (a : f(x)) | x \in e\}.$$

Besides these, we need the regular relational algebra operators ($\sigma, \times, \Pi, \rho, \cup, \cap, \setminus$). Using these operators, we can translate SQL queries into relational algebra.

In the following we will often have to compare sets of attributes. As a shorthand notation, we define the attribute comparison operator $=_A$ as

$$t_1 =_A t_2 \quad := \quad \forall_{a \in A} : t_1.a = t_2.a.$$

Note that unless indicated otherwise this operator has *is* semantics, i.e., it compares NULL values as equal.

3 Unnesting

The algebraic representation of a query with correlated subqueries (initially) results in a dependent join, i.e., an expression of the form

$$T_1 \blacktriangleright\!\!\bowtie_p T_2.$$

As already mentioned, these dependent joins are very unfortunate from a performance perspective, and we want to eliminate them. Fundamentally, we manipulate the algebraic expression until the right hand side no longer depends on the left hand side, and thus the dependent join can be transformed into a regular join. We achieve this using two techniques that we will discuss in the following. First, we try a *simple* unnesting, that handles cases where dependencies are created just for syntactic reasons. If that is not sufficient to unnest the query, we use the *general* unnesting framework that can handle arbitrary complex queries.

3.1 Simple Unnesting

Sometimes queries contain correlated subqueries just because they are simpler to formulate in SQL. An example for that is TPC-H Query 21, which contains a construct similar to the fragment

```
select ...
from   lineitem l1 ...
where  exists (select *
               from lineitem l2
               where l2.l_orderkey = l1.l_orderkey)
...
```

This is translated into an algebra expression of the form

$$l_1 \blacktriangleright (\sigma_{l_1.okey=l_2.okey}(l_2))$$

It is easy to see that this fragment can be unnested by moving the dependent predicate up the tree, transforming the dependent join into a regular join:

$$l_1 \ltimes_{l_1.okey=l_2.okey} (l_2)$$

In general the simple unnesting phase moves all dependent predicates up the algebra tree as far as possible, potentially beyond joins, selections, group by, etc., until it reaches a point where all its attributes are available from the input. If this happens the dependent join can be transformed into a regular join, as shown by the equivalence explained above. Note that this predicate pull-up happens purely for decorreleation reasons. Further optimization steps might push (parts of) the predicate back down again to filter tuples early on.

3.2 General Unnesting

Predicate movement is very easy to implement and already sufficient to handle frequently occuring simply nested queries. Therefore we try it first, but for the general case we need a more complex approach: First, we translate the dependent join into a "nicer" dependent join (i.e., one that is easier to manipulate), and second, we will push the new dependent join down into the query until we can transform it into a regular join.

Thus, in the first step, we use the following equivalence

$$T_1 \bowtie_p T_2 \quad \equiv \quad T_1 \bowtie_{p \wedge T_1 =_{\mathcal{A}(D)} D} (D \bowtie T_2)$$

where $D := \Pi_{\mathcal{F}(T_2) \cap \mathcal{A}(T_1)}(T_1)$.

At a first glance this transformation did not improve the query plan much, as we have replaced one dependent join by a regular join and another dependent join. However, at a second glance this transformation is very helpful: In the original expression, we had to evaluate T_2 for every tuple of T_1, which could be millions. Therefore, in the second expression, we first compute the domain D of all variable bindings, evaluate T_2 only once for every distinct variable binding, and then use a regular join to match the results to the original T_1 value. If there are a lot of duplicates, this already greatly reduces the number of invocations of T_2.

This benefit can be illustrated by considering our first example query for determining the best exam(s) for every student. The straightforward evaluation computes the student's best grade for every exam he or she has ever taken, i.e.:

$$\sigma_{e.\mathrm{grade}=m}((\mathrm{students}\ s \bowtie_{s.id=e.sid} \mathrm{exams}\ e) \bowtie \\ (\Gamma_{\emptyset;m:min(e2.\mathrm{grade})}(\sigma_{s.\mathrm{id}=e2.\mathrm{sid}}\ \mathrm{exams}\ e2)))$$

The equivalence rule allows to restrict the computation of the best grades to each student – instead of computing it redundantly for each (student, exam)-pair. Thus, the dependent join is executed on the projection of the students' id only, i.e.:

$$\ldots \Pi_{d.id:s.id}((\mathrm{students}\ s \bowtie_{s.id=e.sid} \mathrm{exams}\ e) \bowtie \\ (\Gamma_{\emptyset;m:min(e2.\mathrm{grade})}(\sigma_{d.\mathrm{id}=e2.\mathrm{sid}}\ \mathrm{exams}\ e2)))$$

The application of the "Push-Down"-rule for our example query is shown in Figure 1 where the entire query evaluation plan is graphically depicted. In a way, this constitutes a side-ways information passing from the outer (left) join argument to the inner (right) argument in order to eliminate redundancy in the evaluation. Therefore, it is important to implement the projection in the true, duplicate-eliminating semantics and not in the duplicate-preserving multi-set semantics of SQL.

Even more importantly, we have transformed a generic dependent join into a dependent join of a *set* (i.e., a relation without duplicates). Knowing that D contains no duplicates

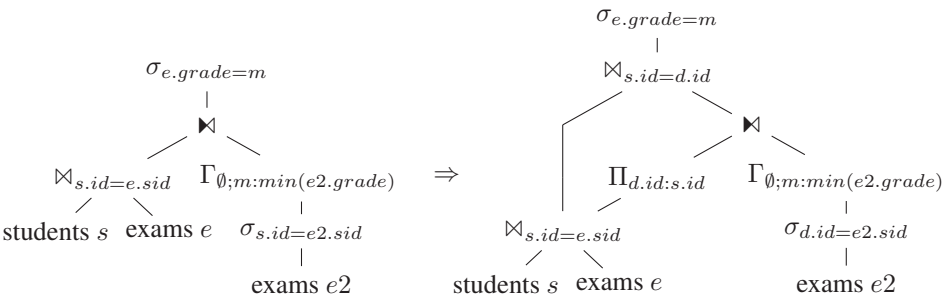


Figure 1: Example Application of Dependent Join "Push-Down"

helps in moving the dependent join further down into the query. In the following we will assume that any relation named D is duplicate free, and in the following equivalences we only consider dependent joins where the left hand side is a set. However, we emphasize that this optimization technique for nested queries does preserve the SQL multi-set semantics. All duplicates – contained in the base relations as well as generated by the query – are retained in the optimized plans; only the set D that constrains the evaluation work of the nested subquery is duplicate free. If duplicates are to be removed (because of a distinct clause in the query) we can further exploit this by pushing duplicate elimination down into the query evaluation plan.

The ultimate goal of our dependent join push-down is to reach a state where the right hand side no longer depends on the left hand side, i.e.,

$$D \bowtie T \quad \equiv \quad D \bowtie T \quad \text{if} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

In this case we still have to perform a join, but at least we can perform a regular join instead of the highly inefficient dependent join. And, as we will see, we can always reach this state. An even nicer goal would be to reach a state where the resulting regular join can be substituted by existing attributes, eliminating the join altogether. We will discuss that in Section 4.

Having explained the start and the goal of our dependent join push down, we now look at individual operators. For selections, a push-down is very simple:

$$D \bowtie \sigma_p(T_2) \quad \equiv \quad \sigma_p(D \bowtie T_2).$$

This transformation might look unusual, as we usually want to push selections down, but that is besides the point of our unnesting transformation: We first push the dependent join down as far as possible, until it can either be eliminated completely due to substitution, or until it can be transformed into a regular join. Once all dependent joins have been eliminated we can use the regular techniques like selection push-down and join reordering to re-optimize the transformed query.

Pushing a dependent join down another join is more complex, as potentially both sides

could depend upon the dependent join

$$D \bowtie (T_1 \bowtie_p T_2) \equiv \begin{cases} (D \bowtie T_1) \bowtie_p T_2 & : \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \bowtie T_2) & : \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \bowtie T_1) \bowtie_{p \wedge \text{natural join } D} (D \bowtie T_2) & : \text{otherwise.} \end{cases}$$

If the values provided by the dependent join are only required on one side we push it to the corresponding side, otherwise we wave to replicate it in both sides. Note that this push-down rule is overly pessimistic, we can often simplify the parts below the join (see Section 4), but we stick to the basic push-down for now. If we pushed the dependent join to both sides we have to augment the join predicate such that both sides are matched on the D values. Note that the replication is not a performance penalty relative to the original expression, in both cases T_1 and T_2 are evaluated $|D|$ times.

For *outer joins* we always have to replicate the dependent join if the inner side depends on it, as otherwise we cannot keep track of unmatched tuples from the outer side.

$$D \bowtie (T_1 \mathbin{⟕}_p T_2) \equiv \begin{cases} (D \bowtie T_1) \mathbin{⟕}_p T_2 & : \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \bowtie T_1) \mathbin{⟕}_{p \wedge \text{natural join } D} (D \bowtie T_2) & : \text{otherwise.} \end{cases}$$

$$D \bowtie (T_1 \mathbin{⟗}_p T_2) \equiv (D \bowtie T_1) \mathbin{⟗}_{p \wedge \text{natural join } D} (D \bowtie T_2).$$

Similar for *semi join* and *anti join*:

$$D \bowtie (T_1 \ltimes_p T_2) \equiv \begin{cases} (D \bowtie T_1) \ltimes_p T_2 & : \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \bowtie T_1) \ltimes_{p \wedge \text{natural join } D} (D \bowtie T_2) & : \text{otherwise.} \end{cases}$$

$$D \bowtie (T_1 \mathbin{▷}_p T_2) \equiv \begin{cases} (D \bowtie T_1) \mathbin{▷}_p T_2 & : \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \bowtie T_1) \mathbin{▷}_{p \wedge \text{natural join } D} (D \bowtie T_2) & : \text{otherwise.} \end{cases}$$

When pushing the dependent join down a *group-by* operator, the group-operator must preserve all attributes produces by the dependent join

$$D \bowtie (\Gamma_{A;a:f}(T)) \equiv \Gamma_{A \cup \mathcal{A}(D);a:f}(D \bowtie T)$$

Again, this makes use of the fact that D is a set.

The *projection* behaves similar to the group by operator

$$D \bowtie (\Pi_A(T)) \equiv \Pi_{A \cup \mathcal{A}(D)}(D \bowtie T)$$

The only missing operators are the set operations

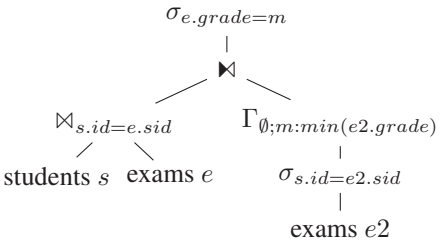


Figure 2: Original Query Q1

$$
\begin{aligned}
D \bowtie (T_1 \cup T_2) &\equiv (D \bowtie T_1) \cup (D \bowtie T_2) \\
D \bowtie (T_1 \cap T_2) &\equiv (D \bowtie T_1) \cap (D \bowtie T_2) \\
D \bowtie (T_1 \setminus T_2) &\equiv (D \bowtie T_1) \setminus (D \bowtie T_2)
\end{aligned}
$$

Using these transformations, each dependent join is either eliminated at some point by substitution, or ends up in front of a base relation, in which case it can be transformed into a non-dependent join. Thus, the dependent join can be eliminated from any query.

One potential concern for this approach could be that D might become very large, as it is the set of all variable bindings for the nested subquery. But fortunately that is not the case. Note that, if the original nested join was $T_1 \bowtie T_2$, then $|D| \leq |T_1|$. Thus, if (after decorrelating the subquery), the top-most join is a hash-join which stores T_1 in a hash table, the memory consumption for that join is at most doubled by computing D. And that is the absolute worst case. If we know that the values from T_1 are duplicate free, for example because they contain a key, we can even avoid materializing D and read the join hash table instead, removing any overhead. On the plus side we have transformed an $O(n^2)$ operation into an (ideally) $O(n)$ operation, which is well worth the memory overhead.

3.3 Optimization of Example Query Q1

As illustrational example, consider the algebraic translation of Query Q1 in Figure 2. It uses a dependent join to compute the nested subquery, and afterwards uses the produced attribute m to check the filter condition.

The subsequent transformations are shown in Figure 3 to 7. First, the top-most dependent join is transformed into a regular join plus a dependent join with the domain of the free

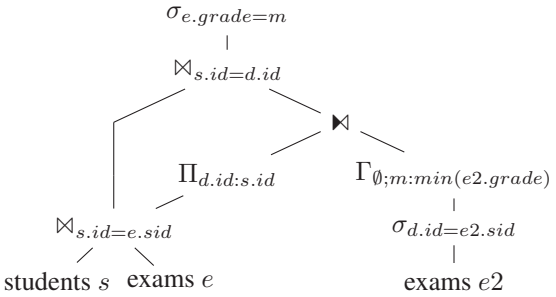


Figure 3: Query Q1, Transformation Step 1

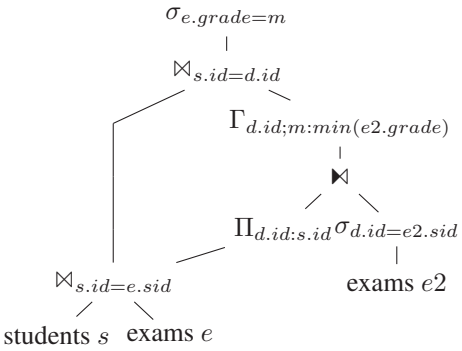


Figure 4: Query Q1, Transformation Step 2

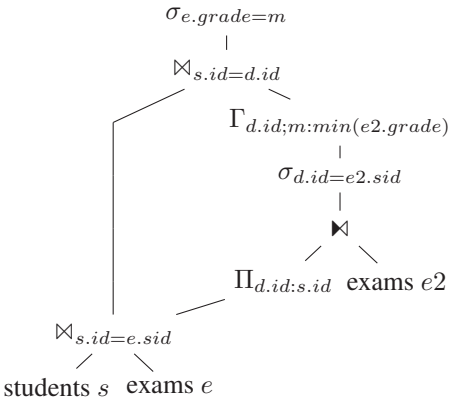


Figure 5: Query Q1, Transformation Step 3

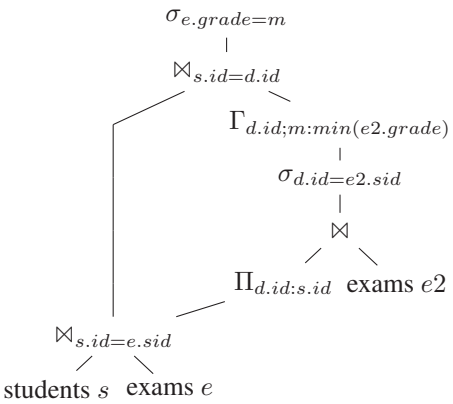


Figure 6: Query Q1, Transformation Step 4

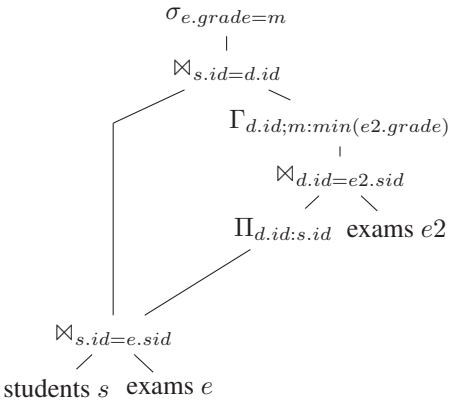


Figure 7: Query Q1, Transformation Step 5 (pushing selections back down)

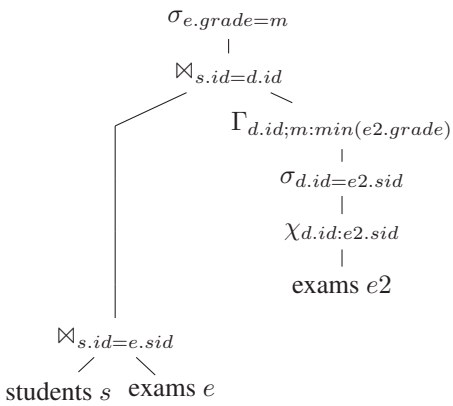


Figure 8: Query Q1, Optional Transformation Step 6 (decoupling both sides)

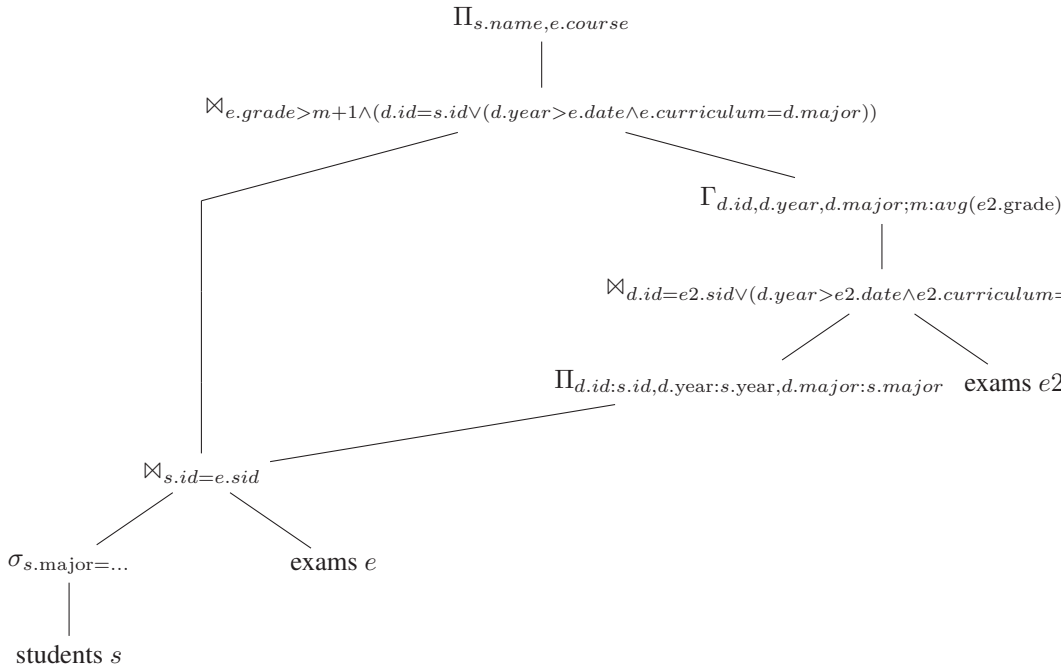


Figure 9: Query Q2, Optimized Form with Sideways Information Passing

variables. In the next step the dependent join is pushed down the group by operator, extending the aggregation attributes as needed. Afterwards, the dependent join is pushed down the selection, ending up in front of a table scan. Now we can transform it into a regular join, as the right hand side is not dependent on the left hand side. Subsequent optimizations will push the predicates down again, introducing a regular join, which results in the plan shown in Figure 7.

In many cases, and also in this example, it is possible to eliminate the join with the domain D altogether: If all attributes of the domain are (equi-)joined with existing attributes, we can instead derive the domain from the existing attributes. In this example, we know that $d.id = e2.sid$ holds after the join, therefore we can replace this by a map operator that substitutes $d.id$ with $e2.sid$, as shown in Figure 8. The nice thing about this substitution is that the parts of the query are completely independent, no trace of the original nesting remains. However, substitution creates a *superset* of the original tuples! At least in general, in this example referential integrity will most likely prevent that. But in general dropping the join with the domain and substituting instead can lead to larger intermediate results, as the filter effect of the join is removed, too. This does not affect the final result, because the filter will still happen at a later stage (at the original dependent join), but it can affect the query runtime. Therefore removing the join is a decision that has to be made by the query optimizer (see Section 4).

Note that some care is needed when trying to remove $\sigma_{d.id=e2.sid}$ after substitution. It would be tempting to simply drop this selection, as $d.id$ is derived from $e2.sid$. But this is only safe if $e2.sid$ is not nullable. If $e2.sid$ can assume NULL values, then the selection must be preserved (and in fact is simply a not-NULL check).

3.4 Optimization of Example Query Q2

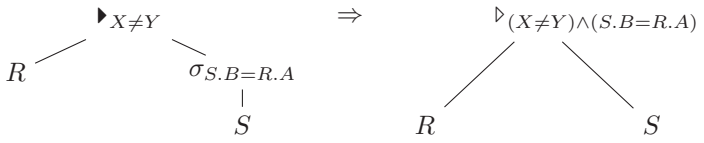
For our motivational query Q2 we illustrate the resulting plan in Figure 9. Note that here D cannot be eliminated, as there is a non-equi join with values from D, which prevents substitution. Here, decoupling the nested subquery evaluation is not possible because the value of $s.year$ is not equi-joinded with $e2.date$. Therefore, the domain of the outer query has to be transferred sideways to the nested query evaluation. However, note that all dependent joins have disappeared and were replaced by efficient regular algebraic operators.

3.5 Anti-Join Example

Let us discuss an example of a dependent anti-join which is used for transforming queries that use an SQL **all**-clause to compare a value against all values derived by a (possibly correlated) subquery. Such a query is formulated below for two abstract relations R : $\{[A, ..., X, ...]\}$ and $S : \{[B, ..., Y, ...]\}$.

```
Q3:  select R.*
     from   R
     where  R.X = all (select S.Y
                       from S
                       where S.B = R.A)
```

Obviously, it is not possible to translate this query into a dependent join; but it is possible to negate the predicate and use a dependent anti-join as shown on the left hand side of the subsequent figure. In the course of the unnesting optimization the dependent anti-join can be transformed into a "normal" anti-join. The resulting query plan is shown on the right-hand side of the subsequent figure.



4 Optimizations

When *simple unnesting* is successful, it completely eliminates any hint of correlations from the query. That is, the resulting algebra expression looks as if the query had been formulated without correlated subqueries. The *general unnesting* case however has to add the projection to compute the domain D, and the join with D, which causes some extra costs. Of course computing D is usually still much preferable to a nested evaluation, as the computation of D and the join with D cause one-time costs, while a nested evaluation results in quadratic runtime. But still, eliminating D completely is tempting, and sometimes possible, as seen in the previous example query Q1 (cf. Figure 8).

In general, we can eliminate D, if we can substitute it with values that already exist in the subtree anyway. This is commonly the case with equi-joins, for example the query contains the expression $D \bowtie_{D.a=R.b} R$, we can learn the possible values of $D.a$ *that can make it to the original dependent join* by inspecting the values of $R.b$. The emphasized part of the statement is important, of course D can contain values that do not exist in R, but these will never find a join partner and will thus never reach the original dependent join. We can therefore ignore them.

To decide about substitution we must first analyze the query tree to find equivalence classes that are induced by the join and filter conditions. For example a filter condition $\sigma_{a=b}$ implies that a and b are in the same equivalence class. We know that in the final result a and b have the same value, we can thus substitute a with b. Computing these equivalence is relatively straight forward. One potential cause for problems would be outer joins, which can cause a and b to not be equal in the example above, but as the top-most join on D is known to be NULL-rejecting this is not an issue here.

After having identified the equivalence classes C, we can decide about a possible substitution as shown below:

$$D \bowtie T \quad \subseteq \quad \chi_{\mathcal{A}(D):B}(T) \text{ if } \exists B \subseteq \mathcal{A}(T) : \mathcal{A}(D) \equiv_C B.$$

Thus, instead of joining with D, we can extend T (using the map operator) and compute the implied attribute value from D by using the equivalent attributes. Note that this only holds because D is a set. Note further that substitution might increase the size of intermediate results, the relationship between the two formulations is not $=$ but $\subseteq$. This cardinality increase is caused by losing the (potential) pruning power of the join with D. Instead of evaluating the formed correlated subtree with every tuple in T that will find a join partner in D, we evaluate with every tuple from T. The tuples that do not have join partners will (only) be eliminated higher up in the tree, when the original dependent join is executed, but the intermediate results can be larger.

Therefore, substitution only pays off if the join with D is unselective. In Query Q1 that will be the case, therefore it is a good idea to use substitution, but in general the query optimization has to compare the costs of both alternatives and chose the cheaper one. For selective joins, it is better to keep D and thus eliminate tuples as early as possible.

5 Evaluation

Unnesting correlated subqueries can lead to nearly arbitrary gains, as it can transform an $O(n^2)$ into an (ideally) $O(n)$ operation. Demonstrating a factor 100 improvement for example would be easy with a carefully constructed query. But the examples and the resulting factors would all be a bit arbitrary. Instead, we therefore first compare the expressiveness of our technique with that of other systems, and then give some performance numbers for TPC-H. All experiments were run on an Intel i7-3930K with 64GB main memory.

We have implemented the unnesting in our HyPer [KN11] system, and compared it to other approaches. First, we studied the unnesting capabilities of several database systems. Out of the commercial systems, SQL Server seems to have the best unnesting engine, which is to be expected based on the publication [GJ01] of the the MS SQL Server team. Unfortunately the licensing terms for SQL Server 2014 forbid publishing runtime numbers, therefore we will only qualitatively describe the result. We also ran experiments on PostgreSQL 9.1, where we are allowed to publish numbers. As test data for our two example queries we generated 1,000 student and 10,000 exam tuples, i.e., 10 exams per student. This is a small data set, but the effects of unnesting are so extreme that we did not want to increase the data set size. (Roughly speaking, the gains of our method grow quadratic with the size of the relations, so we could demonstrate arbitrary gains by increasing the data set size).

Query 1 is relatively simple to unnest. Our HyPer system unnests the query and uses substitution, which result in a runtime of $<$1ms. Without unnesting the query takes 51ms on Hyper. SQL Server 2014 also unnests this query. We are not allowed to report runtimes,

but the query plan is reasonable. PostgreSQL however was not able to unnest even this relatively simple case, which results in a runtime of 1,300ms. And this runtime grows very sharply as the data size grows. If, however, the query would have been formulated decorrelated, as formulated as query $Q1'$ in the Introduction, then PostgreSQL could have executed it in 17ms. This demonstrates how important it is to unnest subqueries.

Query 2 is more difficult to unnest, and we are not aware of any system besides our HyPer system that is able to unnest it. HyPer can execute that query in 42ms (408ms without unnesting). SQL Server 2014 is not able to unnest this query and generates an execution plan with nested loop joins. We are not allowed to report the runtime, but obviously one cannot expect good runtime in a plan with large nested loop joins. PostgreSQL needs 12,099ms for the query, again growing sharply with the data size.

The queries in TPC-H are not that difficult to unnest, and all of the large commercial systems are able to unnest them. This is indeed absolutely essential, as shown by the following performance numbers obtained from our HyPer system that allows to "switch on and off" the unnesting. Even at $SF = 1$, the runtime of Query 4 is 7ms with unnesting, and 157,616ms without. This is a difference of several orders of magnitude, and highlights that unnesting is absolutely essential. Other queries are affected, too: For example Query 17 takes 9ms with unnesting, and 4,664ms without. Of course every vendor makes sure that the well known TPC-H queries are correctly unnested in their system, but these numbers highlight that the performance impact is so large, that a system should be able to unnest arbitrary queries, as proposed in this paper.

6 Related Work

The first, seminal paper on optimization of nested subqueries was published by Won Kim [Kim82]. It provides "recipes" (i.e., transformation rules) for unnesting particular nested query patterns which were integrated in many commercial database systems. A good survey of those techniques was given by Jarke and Koch [JK84]. Werner Kiessling [Kie85] addressed the correctness problems of some of the suggested transformations when empty sets are encountered. For many, but not all, cases he was able to correctly formulate effective unnesting transformations. Thus unnesting remained an active field of research, in particular because of the SQL language extensions that orthogonally allowed nested subqueries in all of the select-from-where-clauses. In the Nineties, Seshadri et al. [SPL96] formulated complex query decorrelationa rules. Dayal's paper [Day87] focuses on efficiently evaluating subqueries utilizing outer joins. The outer joins (also extensively used in our transformation rules) can be optimized using transformation/equivalence rules developed by Galindo-Legaria and Rosenthal [GR97].

Oracle's subquery optimizations are described by Bellamkonda et. al. [BAW$^+$09]. They developed a multitude of subquery optimization techniques, auch as unnesting, group-by merging, common subexpression elimination, join predicate pushdown, join factorization, anti-join formulations of set minus and intersection queries, etc. Many of these techniques are applied as preprocessing transformations in a heuristics pattern-matching way – and, as

it appears, are particularly tuned for optimizing "TPC-H-style" queries. In this paper, we developed a generalized approach that tackles the problem in a uniform way and achieves similar optimization effects as the special-purpose pattern-matching approaches – as our "best of breed" TPC-H performance results reveal [LBKN14].

The Tandem approach for unnesting queries is described by Celis and Zeller [CZ97]. At about the same time, in the context of the Microsoft SL Server, Galindo-Legaria and Joshi [GJ01] introduced the apply-operator, that is similar to our bind join, for "algebrizing" SQL queries with nested subqueries. However, their work fell short of being able to transform all possible nesting patterns: "*Achieving optimality and syntax-independence in this class (subqueries that are removed by introducing additional common subexpressions) requires an understanding of the plan space and mechanisms to generate plans of interest, for queries with common subexpressions, which we believe requires additional research*". We are confident that our paper closes this cited "research gap". Graefe's BTW-paper [Gra03] discusses Microsoft SQL Server's approach to evaluate nested queries that could not be unnested – which was needed when unnesting failed.

A particular query pattern, i.e., scalar subqueries in the presence of disjunctions, was optimized by Brantner, May and Moerkotte [BMM07] using an algebra with bypass operators [KMPS94]. Another important special case of unnesting queries involving large fact tables is addressed by Akinde and Böhlen [AB03]. They argue that outer joins are too costly in OLAP environments with very large fact tables and, instead, propose a generalized multi-dimensional join operator GMDJ.

In the late Nineties, several groups worked on algebra extension to capture nested subqueries. Some of the work was performed in the context of the object-oriented/object relational models [SAB94] that exhibits data nesting and therefore requires flattening and nesting of these structures. One of the first proposals for such an object query algebra was formulated by Cluet and Moerkotte [CM93] and later expanded by Wang, Maier and Shapiro [WMS99]. Cao and Badia [CB05] proposed the explicit use of nested relations to evaluate nested subqueries.

The Virtuoso system [Erl12] tries to address the problem of nested queries at runtime, by evaluating the nested query not once for every outer tuple, but once batch of outer tuples, similar to a block-wise nested loop join, greatly reducing the cost of nested evaluation. During batched execution, each tuple carries a "set number" between operators, maintaining an association between the intermediate results and the vector of inputs of the subquery [Erl14]. While not as powerful as a query-optimizer based solution, this is much more efficient than the standard nested-loop evaluation of nested queries.

Our side-ways information passing optimization to reduce the amount of work incurred by evaluating the nested subquery (tree) resembles the magic set transformations that were described by Seshadri et al [SHP+96] in the context of the IBM DB2 system. Consequently, in our graphics interface (`www.hyper-db.de`) for displaying the query evaluation plans the corresponding operator is called *magic*.

7 Conclusion

Many proposals for unnesting SQL queries have been published in the past. However, so far their implementations fell short of comprehensively cover all possible patterns – as becomes evident by surveying the existing (even recent) literature on optimizing special cases. Also, our analyis of commercial and open source database systems revealed that their optimizations only cover special patterns that (arguably) are the most important use cases for query nesting. In this paper we have developed an algebraic transformation approach that covers all kinds of nested subqueries. The algebraic equivalences allow to completely replace the initially used dependent joins by "regular" join operators. In order to decrease the extent of the nested query evaluation work, side-ways information passing is employed that restricts the independent subquery evaluation plan to those tuple sets that are relevant for the outer query. The approach is fully implemented in our main-memory database system HyPer and can be experimented with via our web interface `www.hyper-db.de` that not only displays the running times of interactive queries but also shows the optimized query evaluation plans. Having integrated the unnesting transformations into our cost-based optimizer ensures that we are never too eager in, for example, decoupling the inner query from the outer query and thereby incurring higher costs – the optimizer will do so only if there is a cost benefit which in many practically relevant cases is dramatic.

Acknowledgements

This work was supported by the German Research Foundation DFG. We acknowldge the helpful comments of the anonymous BTW-reviewers that helped to improve the paper.

References

[AB03] Michael O. Akinde and Michael H. Böhlen. Efficient Computation of Subqueries in Complex OLAP. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 163–174. IEEE Computer Society, 2003.

[BAW+09] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zaït, and Chun Chieh Lin. Enhanced Subquery Optimizations in Oracle. *PVLDB*, 2(2):1366–1377, 2009.

[BMM07] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting Scalar SQL Queries in the Presence of Disjunction. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 46–55, 2007.

[CB05] Bin Cao and Antonio Badia. A Nested Relational Approach to Processing SQL Subqueries. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Con-*

ference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, pages 191–202. ACM, 2005.

[CM93] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 226–242. Springer, 1993.

[CZ97] Pedro Celis and Hansjörg Zeller. Subquery Elimination: A Complete Unnesting Algorithm for an Extended Relational Algebra. In W. A. Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, page 321. IEEE Computer Society, 1997.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 197–208. Morgan Kaufmann, 1987.

[Erl12] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[Erl14] Orri Erling. personal communication, 2014.

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

[GR97] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin Simplification and Reordering for Query Optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[Gra03] Goetz Graefe. Executing Nested Queries. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 58–77. GI, 2003.

[JK84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[Kie85] Werner Kießling. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden.*, pages 241–250. Morgan Kaufmann, 1985.

[Kim82] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[KMPS94] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 336–347. ACM Press, 1994.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[LBKN14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014.

[SAB94] Hennie J. Steenhagen, Peter M. G. Apers, and Henk M. Blanken. Optimization of Nested Queries in a Complex Object Model. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 337–350. Springer, 1994.

[SHP+96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-Based Optimization for Magic: Algebra and Implementation. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 435–446. ACM Press, 1996.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex Query Decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 450–458, 1996.

[WMS99] Quan Wang, David Maier, and Leonard Shapiro. Algebraic unnesting for nested queries. Technical report, Oregon Health & Science University, CSETech. Paper 252, 1999. http://digitalcommons.ohsu.edu/csetech/252.

Optimization of Continuous Queries in Federated Database and Stream Processing Systems

Yuanzhen Ji[1], Zbigniew Jerzak[1], Anisoara Nica[2], Gergor Hackenbroich[1], Christof Fetzer[3]

[1]SAP SE Dresden Germany, [2]SAP SE Waterloo Canada
[3]Systems Engineering Group, TU Dresden, Germany
[1,2]{firstname.lastname}@sap.com, [3]christof.fetzer@tu-dresden.de

Abstract: The constantly increasing number of connected devices and sensors results in increasing volume and velocity of sensor-based streaming data. Traditional approaches for processing high velocity sensor data rely on stream processing engines. However, the increasing complexity of continuous queries executed on top of high velocity data has resulted in growing demand for federated systems composed of data stream processing engines and database engines. One of major challenges for such systems is to devise the optimal query execution plan to maximize the throughput of continuous queries.

In this paper we present a general framework for federated database and stream processing systems, and introduce the design and implementation of a cost-based optimizer for optimizing relational continuous queries in such systems. Our optimizer uses characteristics of continuous queries and source data streams to devise an optimal placement for each operator of a continuous query. This fine level of optimization, combined with the estimation of the feasibility of query plans, allows our optimizer to devise query plans which result in 8 times higher throughput as compared to the baseline approach which uses only stream processing engines. Moreover, our experimental results showed that even for simple queries, a hybrid execution plan can result in 4 times and 1.6 times higher throughput than a pure stream processing engine plan and a pure database engine plan, respectively.

1 Introduction

The increasing amount of connected devices and sensors has led to a surge in the amount, velocity, and value of streaming sensor data. The increasing value of information carried by the streaming sensor data motivates the need to combine and query such data streams. Traditional approaches for processing streaming sensor data rely on stream processing engines (SPE) using continuous queries. A continuous query is issued once and is executed constantly over the data streams, returning a continuous stream of query results.

Existing SPEs are built either from scratch ([ACc+03, KS04, Esp, Pro], etc.) or on top of existing database systems [CH10, FKC+09, LGI09]. Despite this fact, they show limitations in processing certain types of complex continuous queries when compared to modern databases, such as in-memory column stores [Ji13]. In addition, before introducing SPEs, most of today's enterprises already have database systems in place for data persistence and on-demand analytical processing. Hence, the co-existence of a SPE and a DBMS can be found in many real-world setups. Following the philosophy that "no one size fits all",

and aiming to explore the potential of such SPE-database setup, in this paper, we propose to federate the SPE and the database engine for joint execution of continuous queries to achieve performance which cannot be matched by either engine alone. By "federate", we mainly mean outsourcing certain fragments of a continuous query from the SPE to the database engine when the outsourcing can lead to better performance; the federated system, however, supports queries that access both streaming data and stored data as well.

One major challenge of such systems is to find the optimal execution plan for a given continuous query. Existing federated database and stream processing systems either have no federated optimizer at all [BCD+10], or choose the most suitable system for the entire query [LHB13]. Moreover, none of them have considered the *feasibility* property of execution plans of continuous queries, which describes the capability of a plan to keep up with the data arrival rate [AN04]. Finally, the heterogeneity between the underlying SPE and the database engine causes the *non-additivity* of the query execution cost [DH02]. Specifically, the non-additive execution cost means that the cost of executing two consecutive operators in the database engine is not necessarily higher than the cost of executing only the first operator in the database engine. This non-additivity makes it difficult for a query optimizer to make pruning decisions during plan enumeration. Existing solutions used in traditional database systems for handling non-additivity must be extended to consider the feasibility property of plans of continuous queries.

Our major contributions in this paper is the design and implementation of a static cost-based optimizer for optimizing relational continuous queries in a federated database and stream processing system. Our optimizer fully exploits the potential of distributed execution of continuous queries across a SPE and a database engine. Using characteristics of queries and data streams, our optimizer determines an optimal placement for each operator in a continuous query, taking into account the feasibility of query plans and the non-additivity of the query execution cost caused by the federation. To reduce the search space of query plans, we adopt the two-phase optimization strategy [HS91], which is widely used in federated or parallel database systems, as well as systems with heterogeneous multicore architectures (e.g., [HLY+09]). In Phase-One, an optimal logical query plan is produced; in Phase-Two, placement decisions for all operators in the chosen logical plan are made. We further exploit plan-pruning opportunity in Phase-Two based on the study of the cost characteristics of operators placed on the two different engines, thereby reducing the search space further.

As a proof of concept, we federate a commercial stream processing engine—SAP ESP [Pro], and a columnar in-memory database system (IMDB)—SAP HANA [SFGL13], for joint execution of continuous queries. We have implemented the proposed optimization approach by directly extending the optimizer of the IMDB. We experimentally demonstrate that our fine level of optimization, combined with the estimation of the feasibility of query plans, can devise query plans which result in up to 8 times higher throughput when compared to the baseline—the pure SPE-based execution. Our experimental results showed that even for simple queries, the optimizer can derive non-obvious decisions which result in up to 4 times higher throughput when compared to the pure SPE-based execution and up to 1.6 times higher throughput when compared to the pure IMDB-based execution. Our experimental results confirm the superiority and the necessity of a federated optimizer for

continuous queries working at the operator level.

Note that as pointed in [AN04], static query optimization is a valid approach when the characteristics of input streams change slowly or the pattern of change is predictable, which is often observed in data streams originating from sensors with fixed reporting frequencies. Before moving on to a dynamic optimization solution, we must first understand what can be achieved by doing static optimization for continuous queries in a federated database and stream processing system, which is the goal of this paper.

The remainder of this paper is organized as follows. Following the background introduction in Section 2, Section 3 gives an overview of continuous query execution in our prototype federated database and stream processing system. Section 4 defines the query optimization objective in such federated systems. Section 5 drills down to the cost model adopted in our optimizer, followed by descriptions of our two-phase optimization approach in Section 6. Section 7 presents the pruning strategy applied in the second phase of the optimization. In section 8 we experimentally study the effectiveness of our optimizer. We discuss related work in Section 9 and conclude in Section 10.

2 Background

This section presents the semantics of continuous queries adopted in our work (Section 2.1), as well as basics about the pipelined execution model (Section 2.2).

2.1 Continuous Query Semantics

Although a few studies, such as [ABW06, KS09, BDD+10], have tried to offer clean semantic models for continuous queries executed over data streams, to date, there is no established standards. In our work, we adopt the abstract semantics defined in [ABW06], which is based on two data types, *stream*s and *time-varying relation*s, and three classes of query operators. Assuming a discrete and ordered time domain $\mathcal{T}$, streams and time-varying relations are defined as follows:

Stream A stream S is a possibly infinite bag of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the timestamp of s.

Time-varying Relation A time-varying relation R is a mapping from $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to the schema of R. In the following, we call time-varying relations as relations for short wherever the context of stream processing is clear.

The three classes of query operators are the *stream-to-relation* (S2R) operators, which produce one relation from one stream; *relation-to-relation* (R2R) operators, which produce one relation from one or more relations; and *relation-to-stream* (R2S) operators, which produce one stream from a relation. The most typical S2R operator is the *window* operator. There are various types of window operators [ABW06, PS06]. In this paper, we focus on time-based and tuple-based sliding windows. R2R operators are straightforward counterparts of relational operators in conventional database systems. We focus on selection, projection, equi-join, and aggregation in this paper. Without loss of generality, we assume that each R2R operator has at most two input relations; a multi-way join is treated as a sequence of two-way joins. We adopt the following semantics for sliding-window aggregations: aggregation results are produced at each slide of the window.

Table 1: Notations for representing logical and physical plans of continuous queries.

Notation	Description
$T = (\mathcal{O}, \mathcal{E})$	A logical plan of a continuous query CQ
$O_i \in \mathcal{O}$	A logical operator in T
$e_{ij} \in \mathcal{E}$	Data flow from O_i to O_j
$\mathcal{P}(T) = (\mathcal{O}', \mathcal{E}', \mathcal{M})$	A physical plan of T
$O^{spe} \in \mathcal{O}'$	A basic physical operator running in the SPE
$O^{db} \in \mathcal{O}'$	A migration candidate (composite operator) running in the database
$e'_{ij} \in \mathcal{E}'$	Data flow from O_i^x to O_j^y, $x, y \in \{spe, db\}$
$\mathcal{M}(O^x)$	The set of logical operators in T that O^x maps to, $x \in \{spe, db\}$

We have decided to adopt the above described continuous query semantics, because time-varying relations and R2R operators have straightforward semantic mapping to conventional relations and query operators in database systems, respectively. Hence, it provides a sound semantics foundation for federated execution of continuous queries.

2.2 Pipelined Execution

We consider the pipelined query execution model [Gra93], which is adopted by most existing SPEs (e.g., STREAM [ABW06], Aurora [ACc+03]) to adapt to the "push" characteristic of data streams. With pipelined execution, query operators are organized into series of producer-consumer pairs that are connected via a buffering mechanism, e.g., data queues. The producer and the consumer can run in parallel, embodying the so-called *pipelined parallelism* [HM94]. Pipelined execution allows exploiting the power of modern multiprocessor machines to accelerate data processing.

We model pipelined relationships among operators in a continuous query CQ with a directed tree, denoted as $T = (\mathcal{O}, \mathcal{E})$. A node $O_i \in \mathcal{O}$ represents a query operator and an edge $e_{ij} \in \mathcal{E}$ represents the data flow from node O_i to O_j. Similar to [ABW06, KS09], we adopt notions used in conventional database systems, and refer to such a tree as a *logical plan* of CQ. Operators in a logical plan are referred to as logical operators. A logical plan of a continuous query may have multiple semantically equivalent alternatives. Notations used for representing continuous queries are summarized in Table 1.

3 Federated Continuous Query Execution

We have built a prototype system which consists of a state-of-the-art SPE (SAP ESP) and a columnar IMDB (SAP HANA). In this section, we give an overview of continuous query execution in our system.

Determined by the semantic mapping between continuous queries and SQL queries, given a logical plan T of a query, fragments of T that can potentially be executed in the database engine are sub-trees of T that contain only R2R operators. We call such a sub-tree of T as a *migration candidate*. A composition of several R2R operators produces one relation from one or more relations, and hence can be regarded as a R2R operator as well (see Section 2.1). We regard each migration candidate as a *composite R2R operator*. A migration candidate can be translated into a SQL query and executed in the database engine. Particularly, base relations involved in the SQL query map to the input relations of the migration candidate; the result of the SQL query maps to the output relation of the migration candidate.

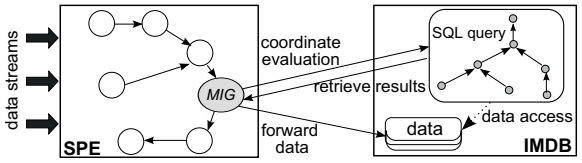


Figure 1: Execution of continuous queries across a SPE and an IMDB.

Figure 1 illustrates how a continuous query is executed across the SPE and the IMDB in our system. The SPE acts as the gateway of external data streams. Federated query execution involves data transfer between the SPE and the database engine. Specifically, for each migration candidate placed in the database engine, we need to transfer relevant input data from the SPE to the database engine; and transfer execution results from the database engine back to the SPE.

To retain the original query semantics, the SQL query corresponding to a migration candidate must be re-executed in response to changes in the input relations of the migration candidate. To coordinate the data transfer between the two engines and the re-execution of the corresponding SQL query, we introduce a new operator *MIG* into the SPE. A *MIG* operator acts as a wrapper of a migration candidate executed in the IMDB. It controls the data transfer between the two engines and hides the execution specifics within the IMDB from the SPE. In a parallel environment, *MIG* operators run in parallel with other query operators in the SPE. However, from the SPE's perspective, each migration candidate wrapped by a *MIG* operator is a black-box, and the original pipelined relationships among operators in the migration candidate is no longer visible.

Execution Plan Representation. Given a logical plan $T = (\mathcal{O}, \mathcal{E})$ of a continuous query, we denote an execution plan of T as $\mathcal{P}(T) = (\mathcal{O}', \mathcal{E}', \mathcal{M})$. $O_i^x \in \mathcal{O}'$ represents a physical operator in the execution plan, where $x \in \{spe, db\}$. Specifically, O_i^{spe} represents a basic query operator (selection, join, etc.) placed in the SPE, and O_i^{db} represents a migration candidate placed in the database engine. For ease of reference, in the remainder of this paper, we refer to a basic query operator placed in the SPE as a *SPE-op* and a composite operator representing a migration candidate placed in the database engine as a *DB-op*. $e'_{ij} \in \mathcal{E}'$ represents the data flow from O_i^x to O_j^y ($x, y \in \{spe, db\}$). Finally, $\mathcal{M}$ defines a mapping from $\mathcal{O}$ to $\mathcal{O}'$. For each $O^x \in \mathcal{O}'$, $\mathcal{M}(O^x)$ defines the subset of $\mathcal{O}$ that O^x maps to. Specifically, $\mathcal{M}(O^{spe})$ is a set containing only one logical operator; $\mathcal{M}(O^{db})$ is a set containing one or more logical operators. Notations used for representing physical plans are summarized in Table 1 as well.

4 The Optimization Objective

A common performance metric for a continuous query executed over data streams is the output rate of the query [GO03]. Therefore, maximizing the query output rate is a widely adopted objective in continuous query optimization [AN04, VN02]. Maximizing the output rate of a query is equivalent to maximizing the amount of input data processed by the query in unit time, which we define as the *query throughput* in this paper. Intuitively, an execution plan reaches its maximum throughput when it can keep up with the data arrival rate. This capability of keeping up with the data arrival rate is defined as the *feasibility* of the plan [AN04]. A continuous query is a *feasible query* if it has at least one feasible plan.

The optimization objective on the query throughput suggests that a query optimizer should favor feasible plans over infeasible plans for feasible queries, and should pick the plan that can maximize the query throughput for infeasible queries. However, what if a query has multiple feasible plans? It has been shown in [AN04] that given enough resources, all feasible plans of a continuous query have the same throughput. Therefore, in this case, we apply a different optimization objective—that is, minimizing the total resource utilization of the query. The motivation behind is that intuitively, the less resources each query consumes, the more number of queries that a system can execute concurrently. In summary, our optimization objective is as follows:

- For feasible queries, find the feasible execution plan which has the least resource utilization.
- For infeasible queries, find the plan which has the maximum the query throughput.

Generally, given two execution plans of a continuous query, possible situations faced by an optimizer, and the respective appropriate optimization decision are the following:

- Situation 1: One plan is feasible and the other is infeasible. $\rightarrow$ Choose the feasible plan.
- Situation 2: Both plans are feasible. $\rightarrow$ Choose the one with less resource utilization.
- Situation 3: Both plans are infeasible. $\rightarrow$ Choose the one with higher throughput.

Discussion. Ayad et al. [AN04] adjust the above optimization objectives to incorporate the influence of load shedding. They insert load shedding operators into plans of an infeasible query, thereby turning all infeasible plans into feasible ones. In this paper, we focus on discussing continuous query optimization in a federated environment and do not consider applying load shedding for infeasible queries.

5 The Cost Model

To achieve the optimization objective described in the previous section, we propose a cost-based optimizer. Without loss of generality, we consider continuous queries whose logical plans have window operators appear only as leaf nodes and R2S operators appear only as root nodes. Note that a query with window or R2S operators appearing as internal nodes can always be split into a set of sub-queries, with the logical plan of each sub-query satisfying the above condition. We also assume a highly parallel environment with abundant memory for query execution. Hence, operators are fully pipelined and do not time-share CPU resources.

We assume that data from source streams arrive at a relatively stable rate. The data rate λ_{S_i} of each source stream S_i in a plan $\mathcal{P}$ defines how much data from S_i should be processed by $\mathcal{P}$ within unit time. We refer to data arrived from all source streams in a plan within unit time as the *unit-time source arrivals*. We further define the amount of data that an operator produces as a result of the unit-time source arrivals as the *source-driven output-size* of the operator, denoted by λ_{out}. Note that (1) the source-driven output-size of an operator is the amount of data produced by the operator as a result of unit-time source-arrivals, but is not the amount of data generated by the operator within unit time, which is also know as the output rate; (2) the source-driven output-size of an operator O_i is used as the *source-driven input-size* (denoted by λ_{in}) by its direct downstream operator O_j to estimate the source-driven output-size of O_j.

Given data rates of all source streams involved in a query, the source-driven output-size λ_{out} of each operator can be estimated in a bottom-up way. In this paper, we adapt the method proposed in [AN04] under our query semantics model (see Section 2.1) to estimate λ_{out} of window-based selection, projection, and join. Specifically, for a selection or a projection with selectivity f[1], its source-driven output-size is

$$\lambda_{out} = f\lambda_{in}. \tag{1}$$

For a join operator, suppose that the size of its left input relation is W_L, the size of the right input relation is W_R, and the selectivities relative to the left and the right relations are f_L and f_R, respectively. Its source-driven output-size can be estimated by Eq. (2). The size of a relation is defined as the number of tuples contained in the relation, which can be estimated in a bottom-up way as described in [AN04].

$$\lambda_{out} = \lambda_{in_L} f_R W_R + \lambda_{in_R} f_L W_L \tag{2}$$

Recall that we define aggregate operators to produce results at each slide of the upstream window (see Section 2.1). For a time-based sliding window, if the slide size is β time units, then on average the unit-time sliding frequency, denoted as l, is $1/\beta$. For a tuple-based sliding window whose slide size is β tuples, the sliding frequency depends on the data rate of the source stream as well, and is estimated as $l = \lambda_s/\beta$. Suppose that the average number of result groups, as determined by the associated grouping predicate of the aggregate operator, is g. We estimate the source-driven output-size of an aggregate operator as

$$\lambda_{out} = lg. \tag{3}$$

5.1 Operator Cost

Having introduced the estimation of source-driven input/output-sizes of operators, we are ready to estimate costs of physical operators in an execution plan. Each tuple arriving at an operator requires some processing effort from the operator. We define the average time that an operator O_j^x requires to process a single tuple from a direct upstream operator O_i^x as the *unit processing cost of O_j^x for O_i^x*, denoted by c_{ji}, or simply c_j if O_j^x has only one direct upstream operator. For an operator O_j^x with k upstream operators, we define the total cost of O_j^x caused by unit-time source-arrivals as the *source-driven input processing cost*, and denote it by u_j. We estimate u_j as

$$u_j = \sum_{i=1}^{k} \lambda_i c_{ji}. \tag{4}$$

To keep up with the data arrival rate, the time needed to process a single tuple by each operator in a pipeline must be shorter than the average data arrival interval at the operator. In other words, the constraint $\sum_{i=1}^{k} \lambda_i c_{ji} \leq 1$, namely $u_j \leq 1$, must hold [AN04, VN02]. An operator that cannot meet the this constraint is the *bottleneck* of the pipeline.

Cost of *SPE-op*s. The cost estimation method described above can be used directly to estimate costs of *SPE-op*s in an execution plan. The unit processing cost c of a specific *SPE-op* depends on the type and the physical implementation of the operator.

[1]The selectivity of a projection is 1.

Cost of *DB-op*s. In contrast to a *SPE-op*, which maps to a single logical operator, a *DB-op* maps to one or more logical operators and is evaluated as one single SQL query. Hence, the unit processing cost of a *DB-op* is practically the execution cost of the corresponding SQL query. Moreover, each time when a *DB-op* is executed, we need to transfer the relevant input data from the SPE, and the execution results back to the SPE (see Section 3). The costs of inter-engine data transfer must be taken into account as well. In summary, the unit processing cost of a *DB-op* consists of three parts: the cost of transferring relevant input data from the SPE to the database, the cost of evaluating the SQL query, and the cost of transferring the SQL query results back to the SPE. In our prototype system, we extended and tuned the built-in cost model of the IMDB to estimate the cost of *DB-op*s.

5.2 Execution Plan Cost

Based on the cost estimation for individual operators described in Section 5.1, we now introduce the cost model for a complete execution plan.

Corresponding to the optimization objectives defined in Section 4, we define the cost of an execution plan $\mathcal{P}$ with m operators, denoted by $C_u(\mathcal{P})$, as a two dimensional vector consisting of two cost metrics: the *bottleneck cost* $C_b(\mathcal{P})$ and the *total utilization cost* $C_u(\mathcal{P})$; namely, $C(\mathcal{P}) = \langle C_b(\mathcal{P}), C_u(\mathcal{P})\rangle$. $C_b(\mathcal{P})$ and $C_u(\mathcal{P})$ are computed as follows:

$$C_b(\mathcal{P}) = max\{u_j^x : j \in [1, m]\}. \tag{5}$$

$$C_u(\mathcal{P}) = \sum_{j=1}^{m} u_j^x \tag{6}$$

Note that here the "bottleneck" refers to the operator with the highest source-driven input processing cost in the plan. We use the bottleneck cost to check the feasibility of a plan. Moreover, for infeasible plans of a query, a higher bottleneck cost implies that the plan can handle fewer input data per unit time; therefore, we also use the bottleneck cost as an indicator of the throughput of an infeasible plan. The total utilization cost estimates the total amount of resources required by the plan to process unit-time source arrivals.

Based on the above cost metrics for execution plans, we define the *optimal plan* of a given continuous query as follows:

Definition 1. *For a continuous query CQ, an execution plan $\mathcal{P}$ is an optimal plan of CQ, iff for any other plan $\mathcal{P}'$ of CQ, one of the following conditions is satisfied:*

Condition 1°: $C_b(\mathcal{P}) \leq 1 < C_b(\mathcal{P}')$
Condition 2°: $C_b(\mathcal{P}) \leq 1$, $C_b(\mathcal{P}') \leq 1$, *and* $C_u(\mathcal{P}) \leq C_u(\mathcal{P}')$
Condition 3°: $1 < C_b(\mathcal{P}) \leq C_b(\mathcal{P}')$

Each condition in Definition 1 applies in a specific situation described in Section 4. Condition 1° is applied when $\mathcal{P}$ is feasible and $\mathcal{P}'$ is infeasible; Condition 2° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are feasible; and Condition 3° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are infeasible.

6 Two-Phase Optimization

In principle, a R2R operator of a query can be executed either in the SPE or in the database engine. However, the placement decision for the operator does not influence its pipelined relationships with its upstream and downstream operators. Consequently, the options of

the execution engine for an operator can be treated as physical implementation alternatives of the operator [BCE+05], thereby allows integrating the selection of the execution engine for operators into the physical plan enumeration phase of a query optimizer.

A continuous query could have a large number of semantically equivalent logical plans due to, for instance, different join ordering possibilities. Even for an individual logical plan T with n R2R operators, there are in total 2^n possible execution plans for T. Due to the large search space of execution plans, exhaustive search for the optimal plan is too expensive. In this paper, following the idea applied in many existing federated, distributed, or parallel database systems, we adopt a *two-phase* optimization approach [HS91]. Specifically, the optimization process is divided into Phase-One, which determines the optimal logical plan for a given query, considering the join ordering and the push-down/up of aggregates, etc.; and Phase-Two, which determines the execution engines of operators in the logical plan picked in Phase-One.

The System R style dynamic programming optimizer [SAC+79] is a widely used query optimizer in existing database systems. It relies on the so-called *principle of optimality* to prune away expensive plans as early as possible. We would like to adopt the System R style optimization approach in our optimizer as well, to find the optimal logical plan in Phase-One. However, to be able to use this approach, we must first show that the principle of optimality holds in the context of continuous query optimization as well; namely, the optimal plan for joining a set of k streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ with another stream S_{k+1} can be obtained by joining stream S_{k+1} with the optimal plan that joins all streams in $\mathcal{S}$.

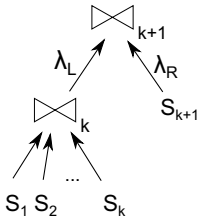


Figure 2: Illustrative logical plan that extends the subplan joining a set of streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ to join with another stream S_{k+1}.

Let us consider the join query in Figure 2. The window operators are skipped for brevity. We denote the optimal plan for joining the set of streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ as $\mathcal{P}_{opt}$. Any suboptimal plan is denoted as $\mathcal{P}_s$. Suppose that the next stream to be joined is S_{k+1}, which incurs λ_R unit-time source-driven arrivals at the new join operator (denoted as $\bowtie_{k+1}$). Note that the total number of join results produced by $\mathcal{P}_{opt}$ as a result of unit-time arrivals from all streams in $\mathcal{S}$ is the same as that produced by $\mathcal{P}_s$. Namely, the source-driven output-sizes of $\bowtie_k$ are identical in all plans that join streams in $\mathcal{S}$. Hence, according to Eq. (4), we can infer that the source-driven input processing cost u of $\bowtie_{k+1}$ is the same in all plans extended from plans for $\bowtie_k$. Denoting the plan extended from $\mathcal{P}_{opt}$ to join with S_{k+1} as $\mathcal{P}'_{opt}$, and the plan extended from $\mathcal{P}_s$ to join with S_{k+1} as $\mathcal{P}'_s$, we now prove that $\mathcal{P}'_{opt}$ is still optimal compared to any $\mathcal{P}'_s$.

Proof Sketch.

Case 1: $\mathcal{P}_{opt}$ is feasible. In this case, a plan $\mathcal{P}_s$ is suboptimal either because it is infeasible (Condition 1° in Definition 1), or because it is feasible as well but has a higher total utilization cost (Condition 2°).

- **Case 1.1:** If $\mathcal{P}_s$ is infeasible, then the plan $\mathcal{P}'_s$ extended from $\mathcal{P}_s$ with $\bowtie_{k+1}$ is still infeasible. Extending $\mathcal{P}_{opt}$ with $\bowtie_{k+1}$ can either leave the resulting plan $\mathcal{P}'_{opt}$ feasible

if $u \leq 1$, or make $\mathcal{P}'_{opt}$ infeasible if $u > 1$. In the former case, it is obvious that $\mathcal{P}'_{opt}$ is better than $\mathcal{P}'_s$. In the later case, we must compare the bottleneck costs of $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$. $C_b(\mathcal{P}'_{opt})$ now equals u. $C_b(\mathcal{P}'_s)$ equals $C_b(\mathcal{P}_s)$ if $u < C_b(\mathcal{P}_s)$, or u if $u \geq C_b(\mathcal{P}_s)$. In either case, we have $1 \leq C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$. Therefore, $\mathcal{P}'_{opt}$ is still optimal (Condition 3°).

- **Case 1.2:** If $\mathcal{P}_s$ is also feasible but has a higher total utilization cost than $\mathcal{P}_{opt}$, then the feasibility of $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ is determined by u in the same way. Specifically, if $u \leq 1$, then both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are feasible. Moreover, $C_u(\mathcal{P}'_s)$ is higher than $C_u(\mathcal{P}'_{opt})$, because $C_u(\mathcal{P}'_s) = C_u(\mathcal{P}_s) + u$, $C_u(\mathcal{P}'_{opt}) = C_u(\mathcal{P}_{opt}) + u$, and $C_u(\mathcal{P}'_s) > C_u(\mathcal{P}'_{opt})$. Therefore, $\mathcal{P}'_{opt}$ is optimal compared to $\mathcal{P}'_s$ according to Condition 1°. If $u > 1$, then both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are infeasible, and we have $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u > 1$. Therefore, $\mathcal{P}'_{opt}$ is still optimal according to Condition 3°.

Case 2: $\mathcal{P}_{opt}$ is infeasible. In this case, $\mathcal{P}_s$ can be suboptimal only when $\mathcal{P}_s$ is infeasible and $1 < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$ (Condition 3°). Plans extended from infeasible plans remain infeasible. Therefore, both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are infeasible. Depending on the value of u, the relationship between $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$ is one of the following cases:

- If $u < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$, then $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}_{opt}) \leq C_b(\mathcal{P}'_s) = C_b(\mathcal{P}_s)$.
- If $C_b(\mathcal{P}_{opt}) \leq u < C_b(\mathcal{P}_s)$, then $C_b(\mathcal{P}'_{opt}) = u < C_b(\mathcal{P}'_s) = C_b(\mathcal{P}_s)$.
- If $C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s) \leq u$, then $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u$.

We can observe that $1 < C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$ (Condition 3°) holds in all three cases. Hence, $\mathcal{P}'_{opt}$ is still optimal. □

Discussion. The above proof shows that the key reasons for the applicability of the principle of optimality are: (1) the source-driven input processing cost u of the new join operator $\bowtie_{k+1}$ is the same in all plans extended from a possible plan that joins streams $S_1, S_2, \ldots, S_k$; (2) u of $\bowtie_{k+1}$ does not change when extending $\bowtie_{k+1}$ to join with other streams.

7 Pruning in Phase-Two

Taking the plan produced in Phase-One as an optimal logical plan, our optimizer determines in Phase-Two the execution engine for each operator in the plan in a bottom-up way. In this section, we describe the pruning strategy used by our optimizer in Phase-Two to further reduce the search space and prove its validity.

By studying the characteristics of the cost of individual *SPE-op*s and *DB-op*s, as well as the influence of their costs on the cost of the entire execution plan, we have observed the following properties of *SPE-op*s: (1) the source-driven input processing cost u of a *SPE-op* O^{spe} is identical in all partial plans rooted at O^{spe}; (2) the source-driven input processing cost of O^{spe} in a partial plan $\mathcal{P}$ rooted at O^{spe} is not changed when $\mathcal{P}$ is further extended. In fact, these two properties are similar to that of the join operators in Figure 2, which suggests that we can apply a similar principle of optimality for pruning. Specificially, to obtain an optimal (partial) plan rooted at a *SPE-op* O^{spe}, it suffices to consider only the optimal partial plans rooted at the direct upstream operators of O^{spe}.

Let us consider the logical plan shown in Figure 3a. Suppose that the current logical operator being enumerated is O_j. Because we adopt a bottom-up enumeration approach,

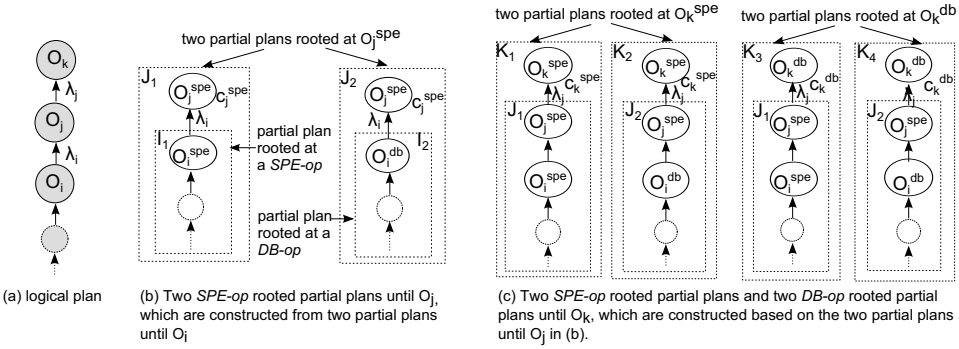


Figure 3: Pruning opportunities when enumerating partial plans rooted at a *SPE-op*.

the enumeration for O_i should have completed. Also suppose that we obtain in total two partial plans until O_i, denoted by I_1 and I_2 (see Figure 3b). I_1 is rooted at a *SPE-op* and I_2 is rooted at a *DB-op*. If we do not consider pruning, we can construct two *SPE-op* rooted partial plans until O_j; one plan extends I_1, denoted by J_1, and the other plan extends I_2, denoted by J_2. We now prove that indeed we need to construct only one *SPE-op* rooted partial plan until O_j, based on the optimal partial plan between I_1 and I_2.

Proof Sketch. This proof consists of two parts. In the first part we show that the optimality relationship between J_1 and J_2 is the same as that between I_1 and I_2. In the second part, we show that for any pair of complete plans $\mathcal{P}_1$ and $\mathcal{P}_2$, the optimality relationship between $\mathcal{P}_1$ and $\mathcal{P}_2$ is the same as that between I_1 and I_2, if $\mathcal{P}_1$ and $\mathcal{P}_2$ differ from each other only by the partial plans until O_j in the way that the partial plan in $\mathcal{P}_1$ is J_1 and in $\mathcal{P}_2$ is J_2.

Part 1: We first show that J_1 is better than J_2 if I_1 is better than I_2. According to Definition 1, there are three possible situations where I_1 can be better than I_2. For each situation, the proof to show that J_1 is better than J_2 is similar to the proof for a specific case discussed in Section 6. Hence, here we provide only references to the corresponding cases in the proof in Section 6.

- **Situation 1:** $C_b(I_1) \leq 1 < C_b(I_2)$, i.e., I_1 is feasible whereas I_2 is infeasible. The proof is similar to that for Case 1.1.
- **Situation 2:** $C_b(I_1) \leq 1$, $C_b(I_2) \leq 1$, and $C_u(I_1)$ ¡ $C_u(I_2)$, i.e., both I_1 and I_2 are feasible. The proof is similar to that for Case 1.2.
- **Situation 3:** $1 < C_b(I_1) \leq C_b(I_2)$, i.e., both I_1 and I_2 are infeasible. The proof is similar to that for Case 2.

The symmetric case that J_2 is better than J_1 if I_2 is better than I_1 can be proved in the same way. Moreover, we can easily extend the proof to show that for an operator O_j with multiple direct upstream operators, the optimal *SPE-op* rooted partial plan until O_j can be constructed from the respective optimal partial plans until each direct upstream operator of O_j.

Part 2: In this part, we show that for a pair of complete plans which are constructed as extensions of J_1 and J_2 respectively, if they differ from each other only by the partial plan J_1 and J_2, then the optimality relationship between them is the same as that between J_1 and J_2. Strictly, we need to show that the optimality is retained along the plan construction

procedure until the root node of the logical plan. However, if we can prove for the direct downstream operator of O_j, which is O_k in Figure 3a, that no matter in which engine O_k is placed, the optimality relationship between the partial plans extended from J_1 and J_2 is the same as the optimality relationship between J_1 and J_2, then we can apply the same reasoning recursively. Therefore, in the following, we only show that for the two partial plan pairs (K_1, K_2) and (K_3, K_4) in Figure 3c, the optimality within each pair is the same as that between J_1 and J_2, and is therefore the same as that between I_1 and I_2.

For the pair (K_1, K_2) where O_k is assigned to the SPE, the same proof in Part 1 can be applied. The proof for the pair (K_3, K_4) is similar. Note that in the partial plans K_3 and K_4, O_k is placed in the database engine, and the source-driven input processing cost u of O_k^{db} is $\lambda_j c_k^{db}$. If the downstream operator of O_k in K_3 and K_4 is placed in the database engine as well, then the two resulting plans, say K_3' and K_4', have a composite operator $O_{k'}^{db}$. The source-driven input processing cost u' of $O_{k'}^{db}$ is $\lambda_j c_{k'}^{db}$. Although u' is different from u, u' is the same in both K_3' and K_4' and therefore does not influence the optimality relationship between K_3' and K_4'. □

Search Space Size. With the above described pruning strategy, for a logical plan with n R2R operators, we get only one *SPE-op* rooted complete execution plan, all the other plans are rooted at a *DB-op*. For logical plans containing only unary operators, we can reduce the search space size from 2^n to $n + 1$. For logical plans containing also binary operators, the search space size depends heavily on the number of binary operators in the tree; because when constructing a *DB-op*-rooted plan at a binary operator, we must consider all possibilities of combining partial plans until the left child of the operator with partial plans until the right child of the operator. In the worst case where all n R2R operators in the plan are binary operators, the logical plan is a complete binary tree. Ignoring window operators at leaf nodes, the height of the tree is $h = \lceil log_2(n + 1) \rceil$. Given the height of a binary tree, we can define the upper bound of the search space size as function of h in a recursive way:

$$f(1) = 2; \quad f(h) = 1 + f(h-1)^2.$$

The complexity of $f(h)$ is $\mathcal{O}(f(h)) = 2^{2^{h-1}}$. By replacing h with $\lceil log_2(n+1) \rceil$, $\mathcal{O}(f(h))$ is approximately $2^{n/2}$, which is exponential. To be able to optimize queries with a large number of binary R2R operators with reasonable time, one solution is to decompose the logical plan produced in Phase-One into multiple subplans, each with a moderate number of binary operators, optimize these subplans in their post order, and construct the final execution plan by combining optimal execution plans of the subplans.

8 Evaluation

In this section, we evaluate the proposed continuous query optimizer from three aspects: the optimization time (Section 8.2), the quality of optimization results (Section 8.3), and the influence of the plan feasibility check on the quality of optimization results (Section 8.4).

8.1 Setup

We implemented the proposed optimization solution by directly extending the SQL optimizer of the columnar IMDB in our prototype. Specifically, we added the cost estimation

for *SPE-op*s, and implemented the proposed two-phase optimization approach. Our system is deployed on a HP Z620 workstation with 24-cores (1.2GHz per core) and 96 GB RAM, running SUSE 11.2.

For our experiments we used the real-world energy consumption data originating from smart plugs deployed in households [JZ14]. Each smart plug is uniquely identified by a combination of a *house id*, a *household id*, and a *plug id*. Each plug has two sensors. One sensor measures the instant power consumption with *Watt* as unit; the other sensor measures the total accumulated power consumption since the start (or reset) of the sensor with *kWh* as unit. Each measurement is represented as a relational tuple. The type of the measurement is indicated by the *property* field in the tuple. Sensors report measurements every 1 second and measurements from all smart plugs are merged into a single data stream. The original rate of this sensor data stream is approximately 2000 tuples/sec. To test with higher data rates, we devised a custom program, which can replay the original sensor data at a configurable speed, simulating a higher report frequency of smart plugs.

We used the following continuous queries ($CQ1$–$CQ6$) to test our federated optimizer:

- $CQ1$**:** For each smart plug, count the number of load measurements in the last 5 minutes whose value is higher than 90% of the maximum load in the last 5 minutes.
- $CQ2$**:** For each smart plug, count the number of load measurements in the last 5 minutes whose value is higher than the average load in the last 5 minutes.
- $CQ3$**:** For each smart plug, compare the maximum and average load within the last 5 minutes with the maximum and average load within the last 1 minute.
- $CQ4$**:** $CQ4$ is similar to $CQ3$ but only compares the average load within the two different time windows.
- $CQ5$**:** For each household, find the maximum total load reported by a single smart plug within the last 5 minutes.
- $CQ6$**:** For each smart plug, compare the average loads within the last 1, 3, and 5 minutes.

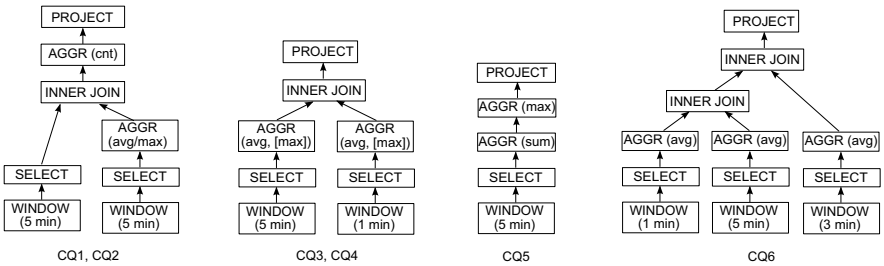


Figure 4: Logical plans of $CQ1$–$CQ6$.

All windows in these queries are time-based sliding windows and slide every 1 second. Figure 4 shows the logical query plans devised by our optimizer. We intentionally included $CQ2$ and $CQ4$ into our test, although they look similar to $CQ1$ and $CQ3$ respectively. The reason is that windowed AVG can be computed incrementally whereas windowed MAX cannot [GHM+07]. Hence, the cost of AVG is normally lower than the cost of MAX in SPEs. We would like to study queries with aggregate operators of different costs.

8.2 Optimization Time

We first tested the efficiency of our optimizer in terms of the optimization time. As mentioned in Section 7, the search space size, thereby the optimization time, is heavily influenced by the number of binary R2R operators in the query. Therefore, in this experiment, we took $CQ4$ as a template and constructed multi-way join queries which compare the average loads of each smart plug within time windows of variant sizes. For instance, a 5-way join query constructed in this way first calculates the average loads of each smart plug within the last 1, 3, 5, 7, and 9 minutes, and then joins these average loads for each smart plug. In this experiment, we did not apply the query decomposition in Phase-Two as discussed in Section 7. For each query, we conducted the optimization 10 times and took the median of the measured optimization times. The results are summarized in Table 2.

Table 2: Optimization time for queries with different numbers of operators.

	2-way join	5-way join	8-way join
Opt. time of Phase-One (ms)	0.9	68.5	100.5
#R2R op. in Phase-One produced logical plan	6	15	24
#plans examined in Phase-Two w/o pruning	64	327168	16777216
#plans examined in Phase-Two with pruning	11	312	8411
Opt. time of Phase-Two with pruning (ms)	12.3	908.6	61335.5
Total opt. time (ms)	13.2	977.1	61436

We can see from the results that with the pruning approach described in Section 7, we significantly reduced the number of plans to be examined in Phase-Two optimization. The results also suggest that in our system, it is reasonable to decompose large logical plans into subplans with 15 operators in Phase-Two. With such decomposition, the logical plan of the 8-way join query produced in Phase-One can be split into two sub-plans, thereby reducing the optimization time from 1 minute to around 2 seconds. Note that we did not provide the optimization time of Phase-Two when the pruning is deactivated, because the experiment would have taken too long and is not meaningful due to the large search space. To be complete, we list the optimization times for $CQ1$–$CQ6$ in Table 3.

Table 3: Optimization time for $CQ1$–$CQ6$.

	$CQ1$	$CQ2$	$CQ3$	$CQ4$	$CQ5$	$CQ6$
Opt. time of Phase-One (ms)	1.3	1.3	0.87	0.86	22.5	5.2
Opt. time of Phase-Two with pruning (ms)	7.7	7.5	11.4	10.8	1.3	58.9
Total opt. time (ms)	9	8.8	12.27	11.66	23.8	64.1

8.3 Effectiveness of the Federated Optimizer

Recall that our optimizer estimates costs of query plans based on data rates of source streams, and finds the optimal plan of a query based on the costs of plans (see Section 5). The data rates of source streams also define the *requested throughput* of a query. For each query in our test, we varied the rate of the sensor data stream from 1000 to 40000 tuples/sec, and asked the optimizer to produce the optimal execution plan for each data rate. For each optimal plan produced by the optimizer, we deployed it in our prototype system, pushed the sensor-data into the system at the corresponding rate, and observed the actual throughput of the plan. The results of this experiment are shown in Figure 5.

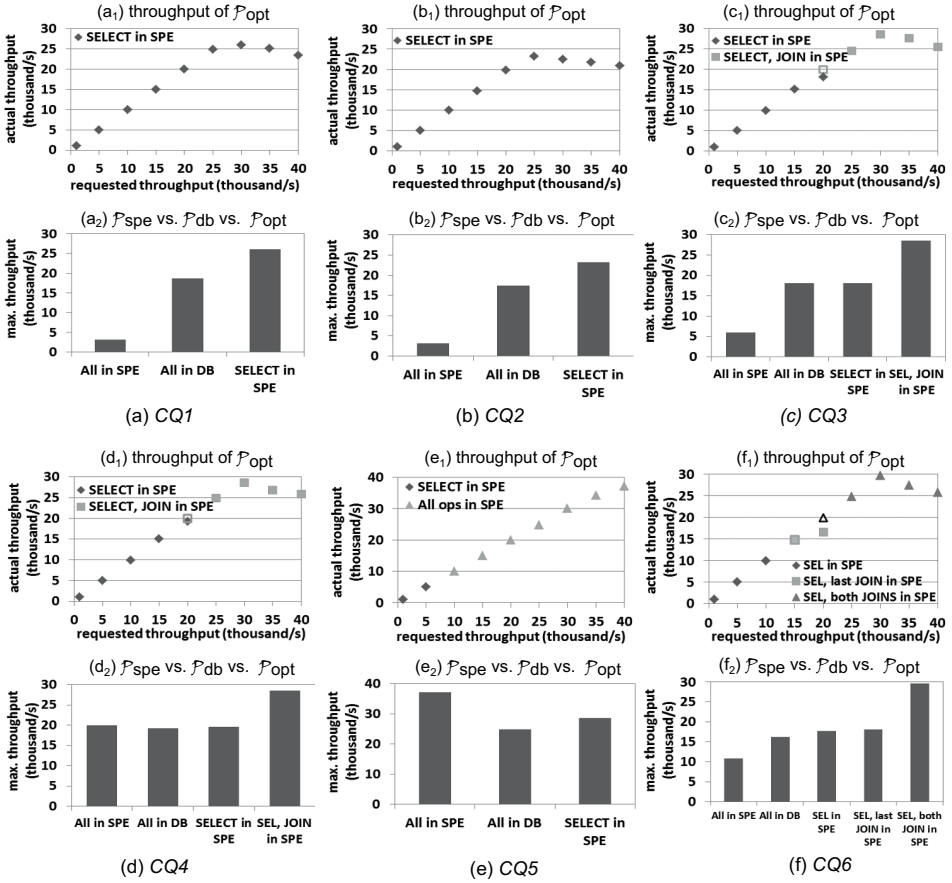


Figure 5: Performance of devised optimal plans for $CQ1$–$CQ6$ at increasing input data rates.

For $CQ1$ and $CQ2$ (see Figure 5a$_1$ and 5b_1), for all examined data rates the optimizer picked the plan which places SELECTs in the SPE and the rest of the operators in the columnar IMDB. The reason for this optimization decision is that $CQ1$ and $CQ2$ compute correlated aggregates, which require scanning the tuples within a time window twice to compute a result. Even for a data rate of 1000 tuples/sec, a 5-minute time window contains 300k tuples. Frequent re-scanning of the window pushes the SPE to its limits. In contrast, the IMDB can compute the correlated aggregate more efficiently, despite the cost of transferring data between the two engines. The SELECTs are placed in the SPE to reduce the amount of data to be transferred to the IMDB.

To verify the superiority of our operator-level optimization approach over a query-level optimization approach, we compared the *maximum throughputs* of the optimal federated plan $\mathcal{P}_{opt}$, the pure SPE plan $\mathcal{P}_{spe}$, and the pure IMDB plan $\mathcal{P}_{db}$ for $CQ1$ and $CQ2$. We see from Figure 5a$_2$ and Figure 5b$_2$ that for both queries, the optimal federated plan can result in about 8 times higher throughput than the pure SPE plan. The maximum throughput of the pure IMDB plan is also lower than the federated plan, because it transfers more data from the SPE to the IMDB, thereby resulting in a higher cost.

For $CQ3$ (see Figure 5c$_1$), the plan which places only SELECTs in the SPE (denoted by $\mathcal{P}_{opt1}$) remains optimal until the data rate reaches 20k tuples/sec. For higher data rates, $\mathcal{P}_{opt1}$ becomes infeasible, and the plan which places both SELECTs and JOIN in the SPE (denoted by $\mathcal{P}_{opt2}$) becomes optimal. Note that when the data rate is below 20k tuples/sec, $\mathcal{P}_{opt2}$ is also feasible; however, it is not picked by the optimizer because it has higher total utilization cost than $\mathcal{P}_{opt1}$. The maximum throughputs shown in Figure 5c$_2$ confirm that $\mathcal{P}_{opt1}$ becomes infeasible at a lower data rate compared to $\mathcal{P}_{opt2}$. When the data rate is 20k tuples/sec, the actual throughput of $\mathcal{P}_{opt1}$ is indeed lower than the requested throughput, which suggests that $\mathcal{P}_{opt1}$ is already infeasible at this date rate, and $\mathcal{P}_{opt2}$ should have been chosen. The throughput of $\mathcal{P}_{opt2}$ at the rate of 20k tuples/sec is indicated by the hollow square in Figure 5c$_1$. This outcome of missing the actual optimal plan is caused by the imperfection of the cost estimation, which we believe is a common issue shared by all cost-based optimizers. However, the difference between the actual throughputs of $\mathcal{P}_{opt1}$ and $\mathcal{P}_{opt2}$ is small, and the optimizer successfully finds the correct optimal plan for all the other examined data rates. For $CQ3$, the federated plan again results in higher throughput than the pure SPE and IMDB plans.

The optimization results for $CQ4$ is similar to that for $CQ3$ (see Figure 5d$_1$). However, the pure SPE plan of $CQ4$ can support much higher data rate than the pure SPE plan of $CQ3$ (see Figure 5d$_2$), which confirms that computing MAX is more expensive than computing AVG in the SPE.

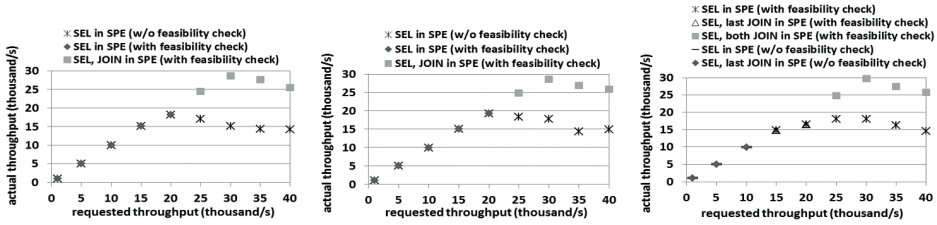
For $CQ5$ (see Figure 5e$_1$), the optimizer picked the plan which places SELECT in the SPE when the data rate is below 10k tuples/sec. For higher data rates, the total utilization cost of this plan becomes higher than that of the pure SPE plan, due to the increasing cost of data transfer between the two engines. As a result, the optimal plan switches to the pure SPE plan. Moreover, unlike for $CQ1$-$CQ4$, the pure SPE plan of $CQ5$ has higher maximum throughput compared to its federated plan alternatives (see Figure 5e$_2$).

$CQ6$ is a 3-way join query. Its optimal plan changed twice as the increase of the data rate (see Figure 5f$_1$). For data rates below 10k tuples/sec, the optimal plan has only SELECTs in the SPE (denoted by $\mathcal{P}_{opt1}$). At higher data rates until 20k tuples/sec, the optimal plan has the second JOIN operator in the SPE as well (denoted by $\mathcal{P}_{opt2}$). For even higher data rates, only the aggregation operators are left in the IMDB (denoted by $\mathcal{P}_{opt3}$). The switch from $\mathcal{P}_{opt1}$ to $\mathcal{P}_{opt2}$ was due to the higher total utilization cost of $\mathcal{P}_{opt1}$, and the switch from $\mathcal{P}_{opt2}$ to $\mathcal{P}_{opt3}$ was due to the infeasibility of $\mathcal{P}_{opt2}$ (Figure 5f$_2$). Similar to the case of $CQ3$ and $CQ3$, the optimizer missed the actual optimal plan at the rate of 25k tuples/sec, as indicated by the hollow triangle in Figure 5f$_1$.

In summary, our federated optimizer performs well with respect to the quality of optimization results. Especially, for each examined query, when the data rate of the source stream is so high that the query becomes infeasible, our optimizer is able to choose the plan which can maximize the query throughput.

8.4 Influence of the Plan Feasibility Check

Last, we studied the influence of the plan feasibility check on the quality of optimization results. To do this, we turned off the feasibility check of query plans in the optimizer, and repeated the tests described in the previous section for all six queries. For each query, we



(a) $CQ3$ (b) $CQ4$ (c) $CQ6$

Figure 6: Throughput of optimal plans devised with and without plan feasibility check.

compared the actual throughputs of the optimal plans devised with and without the plan feasibility check under each examined data rate.

The optimization results for $CQ1$, $CQ2$, and $CQ5$ without plan feasibility check are identical to that with plan feasibility check. However, for $CQ3$ and $CQ4$, without plan feasibility check, the optimizer picked the plan which places only SELECTs in the SPE at all examined data rates. However, this plan is suboptimal than the plan devised with feasibility check when the data rate is above 20k tuples/sec (see Figure 6a and 6b). For $CQ6$, without the feasibility check, the optimizer did not pick the plan which has only aggregate operators in the IMDB when the data rate is above 30k tuples/sec, which indeed has higher throughput (see Figure 6c). These results confirm the necessity of the plan feasibility check in continuous query optimization. It also implies that naive approaches for partial plan pruning without considering the plan feasibility may result in suboptimal plans.

9 Related Work

Leveraging database engines for data stream processing has been studied in a few prior works [CH10, FKC$^+$09, LGI09]. Truviso [FKC$^+$09] integrates the continuous analytics technology into a fully functional database system by executing SQL queries continuously and incrementally over data before storing the data in the database. DataCell [LGI09] is a stream processing engine built on top of MonetDB. Chen et al. [CH10] extend PostgreSQL to support stream processing. This body of work focuses on studying how a database engine can be modified to support stream processing. In contrast, our work aims to make use of the fact that SPEs and modern database systems already co-exist in many real-world setups, and proposes federated optimization and execution of continuous query to leverage the advantages of both types of systems. Our experimental results in Section 8.3 confirm the potential of federated database and stream processing systems. For the same reason, we did not follow the approach of extending the SPE directly with same implementations used in the IMDB, which can avoid the inter-system data transfer and lead to a better query processing performance. However, our query optimization approach can be applied by such extended SPEs as well to determine the best implementation alternative of each query operator from all alternatives available in the system.

MaxStream [BCD$^+$10] is a federated stream processing system which integrates multiple SPEs and databases. The federator layer is built on top of a relational database system. In MaxStream, data streams first pass through the federator layer, where the data are persisted into, or joined with static database tables if needed; subsequently, the streams are forwarded to a specific SPE for processing. However, MaxStream does not have an optimizer

for continuous queries. ASPEN [LMB+10] is a project about integrating and processing distributed stream data sources in sensor devices, traditional PCs, and servers. It has a federated optimizer to assign queries across multiple subsystems. However, the optimizer does not consider the feasibility of continuous query plans, and lacks experimental support for its effectiveness. Cyclops [LHB13] integrates a centralized stream processing system (Esper [Esp]), a distributed stream processing system (Storm [Sto]), and a distributed batch system (Hadoop [Apa]) for executing continuous windowed aggregate queries. Cyclops uses black-box modeling to build cost models. Its optimizer selects the most suitable system for a given continuous query based on the window specification (range and slide) and the data arrival rate. In contrast, our optimizer works at the operator granularity, whose superiority has been confirmed by our experimental results.

Optimization of SQL queries in federated or distributed database systems [BCE+05, DH02, SL90] has been well-studied. However, existing solutions cannot be used directly for federated continuous query optimization, because they do not consider the feasibility of continuous query plans. Optimization of continuous SPJ queries concerning the plan feasibility and query throughput was initially studied in [VN02], and was extended by [AN04], which considers the optimal placement of load shedding operators in infeasible plans when computation resources are insufficient. Cammert et al. [CKSV08] deal with the similar resource management problem, and propose techniques which are based on the adjustment of window sizes and time granularities. Moreover, the cost model in [CKSV08] supports queries containing aggregation operators. However, these works do not consider query optimization in federated systems as described in this paper. There is a large body of work about operator placement in distributed or heterogeneous stream processing environments (e.g., [DLB+11] and works surveyed in [LLS08]). These works normally assume that the pipelined relationships among query operators are already determined, and consider only the placement of operators in the available processing nodes/systems. Furthermore, they do not adopt the feasibility-dependent optimization objective as we do. Nevertheless, studying how to adapt these optimization approaches in our Phase-Two optimization would be an interesting direction for future work.

10 Conclusion

In this paper, we propose a cost-based query optimization approach for federated execution of continuous queries over a SPE and a database system. To fully exploit the potential of hybrid execution of continuous queries, our optimizer works at the operator level and determines the optimal placement for each operator in a query based on the characteristics of the query and involved data streams. Moreover, the optimizer takes into account the feasibility of continuous query plans and the non-additivity of the query execution cost caused by the federation. We experimentally demonstrated the effectiveness of our optimizer in a prototype system composed of a state-of-the-art SPE and a columnar IMDB. Even for simple queries, our optimizer can make non-obvious decisions which result in up to 4 and 1.6 times higher throughput compared to the pure SPE-based execution and the pure IMDB-based execution, respectively. This result confirms that distributed execution of continuous query across SPEs and database engines is viable and promising, worthy of further exploration.

For future work, we plan to relax our assumptions on the static environment and consider

runtime re-optimization in response to changing stream characteristics. Extending the current solution to support multi-query optimization (MQO) is another important direction. Apart from finding common fragments among multiple queries as is done by conventional MQO methods, we envision the sharing of inter-system data transfer channels as a particularly important aspect in the extended solution to save the communicate cost. In addition, in principle, we can add more SPEs or database engines into our system. With more than two engines, the two-phase optimization approach can still be applied; however, the pruning strategy currently applied in Phase-Two needs to be revisited.

References

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[AN04] Ahmed M. Ayad and Jeffrey F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams. In *SIGMOD*, pages 419–430, 2004.

[Apa] Apache Hadoop. `http://hadoop.apache.org/`.

[BCD+10] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Ankush Gupta, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun, and Jin Zhang. A demonstration of the MaxStream federated stream processing system. In *ICDE*, pages 1093–1096, 2010.

[BCE+05] J.A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.-C. Wu. Distributed/heterogeneous query processing in Microsoft SQL server. In *ICDE*, pages 1001–1012, 2005.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *VLDB*, 3(1-2):232–243, September 2010.

[CH10] Qiming Chen and Meichun Hsu. Experience in extending query engine for continuous analytics. In *DaWaK*, pages 190–202, 2010.

[CKSV08] Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE TKDE*, 20(2):230–245, February 2008.

[DH02] Amol V. Deshpande and Joseph M. Hellerstein. Decoupled Query Optimization for Federated Database Systems. In *ICDE*, pages 716–, 2002.

[DLB+11] Michael Daum, Frank Lauterwald, Philipp Baumgärtel, Niko Pollner, and Klaus Meyer-Wegener. Efficient and Cost-aware Operator Placement in Heterogeneous Stream-processing Environments. In *DEBS*, pages 393–394, 2011.

[Esp] Esper. `http://esper.codehaus.org`.

[FKC+09] Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *CIDR*, 2009.

[GHM+07] T.M. Ghanem, M.A. Hammad, M.F. Mokbel, W.G. Aref, and A.K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE TKDE*, 19(1):57–72, 2007.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[HLY+09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[HM94] Waqar Hasan and Rajeev Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *VLDB*, pages 36–47, 1994.

[HS91] Wei Hong and Michael Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, pages 218–225, 1991.

[Ji13] Yuanzhen Ji. Database support for processing complex aggregate queries over data streams. EDBT Workshop, pages 31–37, 2013.

[JZ14] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 Grand Challenge. In *DEBS*, pages 266–269, 2014.

[KS04] Jürgen Krämer and Bernhard Seeger. PIPES: a public infrastructure for processing and exploring streams. In *SIGMOD*, pages 925–926, 2004.

[KS09] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM TODS*, 34(1):4:1–4:49, April 2009.

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. Exploiting the power of relational databases for efficient stream processing. In *EDBT*, pages 323–334, 2009.

[LHB13] Harold Lim, Yuzhang Han, and Shivnath Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.

[LLS08] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing*, 12(6):50–60, November 2008.

[LMB+10] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. SmartCIS: integrating digital and physical environments. *SIGMOD Record*, 39(1):48–53, 2010.

[Pro] SAP Event Stream Processor. `http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html`.

[PS06] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *EDBT*, pages 445–464, 2006.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[SFGL13] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. SAP HANA: The Evolution from a Modern Main-memory Data Platform to an Enterprise Application Platform. *Proc. VLDB Endow.*, 6(11):1184–1185, August 2013.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990.

[Sto] Storm. `http://storm-project.net/`.

[VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

Detection and Recognition

Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Abstract: Although most NoSQL Data Stores are schema-less, information on the structural properties of the persisted data is nevertheless essential during application development. Otherwise, accessing the data becomes simply impractical.
In this paper, we introduce an algorithm for schema extraction that is operating outside of the NoSQL data store. Our method is specifically targeted at semi-structured data persisted in NoSQL stores, e.g., in JSON format. Rather than designing the schema up front, extracting a schema in hindsight can be seen as a reverse-engineering step. Based on the extracted schema information, we propose set of similarity measures that capture the degree of heterogeneity of JSON data and which reveal structural outliers in the data. We evaluate our implementation on two real-life datasets: a database from the Wendelstein 7-X project and Web Performance Data.

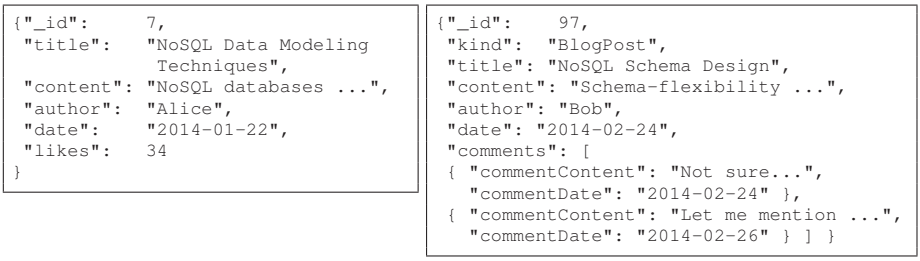
1 Introduction

Most NoSQL data stores do not enforce any interesting structural constraints on the data stored. Instead, the persisted data often has only *implicit* structural information, e.g., in NoSQL document stores that manage collections of JSON documents. Yet when accessing this data programmatically, it is vital to have some reliable notion of its structure, or schema. This holds for traditional application development, as well as when analyzing large scientific data sets: Without knowing the general structure of the data, and in particular its structural outliers, it becomes virtually impossible to do any pragmatic application development or meaningful data analysis. Yet when working with NoSQL data stores, the schema is frequently unknown, either due to rapid schema evolution in agile development, or, as we will show, when a scientific database contains data from different experiments.

NoSQL data stores excel at handling big volumes of data, and schema-less stores in particular are frequently chosen to manage data with a high degree of structural variety.

For instance, let us assume a startup releasing a blogging software. The JSON documents from Figure 1 have been persisted by two different versions of the application. In between the release producing the document in Figure 1(a) and the document in (b), the property *likes* has been dropped, and the functionality to comment on blogs has been added.

With the software-as-a-service model, it is common that applications are re-released into the cloud on a weekly, if not daily basis. In such settings, NoSQL data stores are particu-



(a) First-generation blogpost

(b) Second-generation blogpost

Figure 1: The structure of persisted JSON documents changes due to software- and schema evolution. Schema-free NoSQL data stores allow for heterogeneous documents to co-exist

larly popular back ends for data storage, since their schema-flexibility allows for heterogeneous data to co-exist. After all, most NoSQL data stores do not maintain a data dictionary and thus do not actively manage the schema of the data. This makes it unnecessary to migrate the persisted data with each new release of the software. At the same time, the complexity of handling heterogeneously structured data is building up [KSS14, SKS13].

When developers access heterogeneous data programmatically, it is vital to have some reliable notion of its schema. Let us emphasize on this point for different use cases:

- To build an interactive web application that reads and writes persisted data, e.g., a Java application using object mappers, developers need object mapper class definitions that robustly match blogpost documents from both generations. [SHKS15]
- Many NoSQL data stores provide a declarative query language. To formulate queries, developers need to know which attributes are present (or absent) in persisted objects.
- For OLAP-style data analysis, e.g., in the form of MapReduce jobs, developers also need to know which structure to expect when parsing JSON documents.

Thus, any pragmatic approach to these tasks requires some form of schema description. Figure 2 shows an instance of such a description in *JSON schema* [JSO14]. This specification matches both generations of blogposts in Figure 1, provides type information as well as information on the nesting of properties, and denotes which properties are required.

Out of this motivation, we present a new *schema extraction* approach that is custom-tailored to working with JSON documents. Our algorithm builds upon ideas proposed for extracting DTDs from collections of XML documents [MLN00], but takes into account the particularities of JSON data, such as the unordered nature of JSON documents (unlike XML, where the order of elements plays an essential role) and measures describing the structural heterogeneity of a set of JSON documents. Our algorithm outputs a JSON schema declaration. This declaration can then be the basis for providing convenient programmatic access to the persisted data, e.g.,

- by generating *type provider libraries* that allow convenient access to structured data

```
{  "type": "object",
   "properties": {
      "_id": { "type": "integer" },
      "title": { "type": "string" },
      "content": { "type": "string" },
      ...
      "comments": { "type": "array",
         "items": { "type": "object",
            "properties": {
               "commentContent": { "type": "string" },
               "commentDate": { "type": "string" } }
            "required": ["commentContent", "commentDate"] } } }
   "required": ["title", "content", "author", "date"] }
```

Figure 2: A JSON schema declaration for blogpost data

from within integrated development environments [SBT+12].

- by generating a hierarchy of object mapper class declarations, e.g., JPA [Jav09] or JDO [Jav03] Java class declarations, to be used in application development, or
- to generate validating JSON parsers directly from the JSON schema, in the tradition of parser generators and validating XML parsers.

The extracted JSON schema is an *explicit schema*. Our algorithm uses an internal graph structure that summarizes all structure variants encountered in the data. This allows us to also detect structural *outliers* (patterns that occur only in few data sets and might even be due to a errors during recording of the data). Further, we can determine the *degree of coverage* of the documents, a similarity measure capturing the structural homogeneity of a document collection.

In earlier work, we have proposed a database-external schema management component, a layer used on top of a NoSQL data store [KSS14]. This layer is to preserve the flexibility of the NoSQL data store, while at the same time making it possible to detect and correct structural inconsistencies in the data. The schema extraction algorithm, as presented in this paper, is a vital building block towards this vision.

Structure. This article is organized as follows: The next section gives an overview on different classes of NoSQL data stores. In Section 3, we formalize NoSQL documents and JSON schema, a schema language that is currently under discussion in the community. We present our schema extraction algorithm based on the **S**tructure Identification **G**raph (SG) in Section 4. We detect structural outliers in the data (Section 4.5), and introduce similarity measures in Section 4.6. Since SGs can become too large to fit in memory, we propose an alternative approach that uses a **R**educed Structure Identification **G**raph (RG) in section 5. We conduct our experiments on real-world data. The main results are provided in Section 6. The article concludes with an outlook on how the schema extraction can be implemented as part of a holistic schema management component.

2 NoSQL Data Stores

NoSQL data stores vary hugely in terms of their data and query model, scalability, architecture, and persistence design. The most common and for our context best suited categorization is by data model. Therefore we distinguish *key-value stores*, *document stores*, and *extensible record stores* [Cat10, Tiw11]. Often, extensible record stores are also called *wide column stores* or *column family stores*.

Key-value stores persist pairs of a unique key and an opaque value. There is no concept of schema beyond distinguishing keys and values. Therefore, applications modifying the data have to maintain the structural constraints and therefore also the schema.

Document stores also persist key-value pairs, yet the values are structured as "documents". This term connotes loosely structured sets of name-value pairs, typically in JSON (JavaScript Object Notation) [Ecm13] format or the binary representation BSON, a more type-rich format. Name-value pairs represent the properties of data objects. Names are unique within a document, and name-value pairs are also referred to as key-value pairs.

Documents are hierarchical, so values may not only be scalar values or lists, but even nested documents. Documents within the same document store may differ in their structure, since there is no fixed schema. If a document store is schema-less, documents may effortlessly evolve in structure over time: Properties can be added or removed from a particular document without affecting the remaining documents.

Extensible record stores actually provide a loosely defined schema. Data is stored as records. A schema defines families of properties, and new properties can be added within a property family on a per-record basis. (Properties and property families are often also referred to as *columns* and *column families*.)

Typically, the schema cannot be defined up front and extensible record stores allow the ad-hoc creation of new properties. The Cassandra system [Tiw11, Apa13] is an exception among extensible record stores, since it is much more restrictive regarding schema. Properties are actually defined up front, even with a "create table" statement, and the schema is altered globally with an "alter table" statement.

NoSQL Data Stores in Scope of this Paper. Data in document stores is often represented in JSON format. This format contains implicit structural information and is an interesting starting-point for schema extraction. Therefore, the schema extraction methods we present in this paper primarily focus on JSON-based document stores. Schema extraction can also be applied to extensible record stores due to their concept of explicitly naming properties and property families. The approach presented in this paper can easily be transferred to extensible record stores. Since key-value stores are not aware of schema apart from distinguishing keys and values, we believe they are not in the main focus for schema extraction. However, data in key-value stores may also be stored in JSON or a similar format. Then, schema extraction may also be applied to key-value stores.

3 JSON Documents and JSON Schema

Several NoSQL databases store JSON documents. In JSON the entities are called *objects*. Objects are unordered enumeration of *properties*, consisting of *name/value* pairs [Ecm13]. The available basic data types are *number, string*, and *boolean*. JSON properties can be multi-valued, these structures are then called *arrays*. Arrays are ordered lists of values, written in square brackets and separated by commas.

Schema for NoSQL Data Stores. For declaring a schema and validating data against a schema, we need an explicit schema description. Since we do not want to propose our own language, resort to an already available schema description language: JSON Schema [JSO14] defines structural constraints on sets of JSON objects.

In this article, we do not introduce the complete schema language. Instead, we provide some intuition using a small example in Figure 2. An *object* `blogpost` is represented by the *properties* `title, content, author, date, likes,` and `comments`. These properties are listed, the type of each property is defined. All *required* properties are enumerated, all other properties that are defined in the schema are *optional*. In the schema in Figure 2, the properties `likes` and `comments` are optional. The property `comments` is defined as an array, consisting of objects. Within the objects the properties `commentContent` and `commentDate` occur. Both properties are enumerated in the list of required properties that is given within the definition of the `comments`. The schema in Figure 2 uses the JSON data types `string`, `array`, and `object`. It even contains a hierarchical nesting of objects. In the following, we derive a JSON schema from a collection of JSON documents.

4 Schema Extraction with a Structure Identification Graph (SG)

In this chapter, we introduce our method for schema extraction from a collection of JSON documents. Our approach is related to existing work on DTD extraction from XML documents [MLN00]. Figure 3 states the sub-tasks of our extraction approach.

A graph data structure summarizes the structural information of all persisted documents. The nodes represent JSON properties, nested objects, or arrays, and the edges capture the hierarchical structure of the JSON documents. In particular, nodes and edges are labeled with lineage information, i.e. lists that specify in which JSON documents the structural property is present. The graph structure will be defined in Section 4.2 in detail and the construction of the graph will be described in Section 4.4.

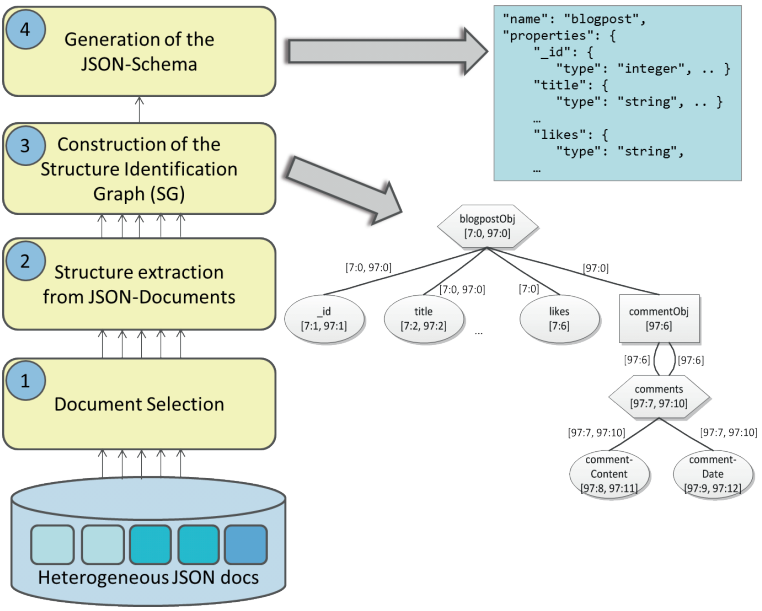


Figure 3: The sub-tasks of schema extraction

4.1 Document Selection

Our schema extraction algorithm derives an explicit schema from a collection of JSON documents. In a preselection step, we choose to run the schema extraction on either the *complete collection* or *selected subsets of JSON documents*.

There are several use cases motivating this document preselection step:

- It may be of interest to divide the collection of JSON documents into groups (distinguished by date, timestamp, or version number), and then extract the *schema for each group or version*. This can reveal the evolution of the schema over time.
- If a split attribute exists that can be applied for *distinguishing the JSON documents of a collection*, then we can build schema for a subset of documents. We can, for instance, distinguish documents from different servers, from different geographical places, or documents that describe different kinds of devices.

In summary, we preselect a NoSQL subset if different kinds of JSON documents are organized in the same collection. In all cases, the preprocessing can be done if a *JSON property or property set* for selecting a subset exists.

4.2 Definition of the Structure Identification Graph (SG)

We now define our internal graph data structure for schema extraction, with node and edge labels that capture the complete schema structure. Our schema extraction algorithm in Section 4.4 is based on this graph. The schema information from *all JSON documents* of a Collection or subset of a Collection is stored in a so-called **S**tructure Identification **G**raph.

Definition 3: A **S**tructure Identification **G**raph $SG = (V, E)$ is a directed graph where

- V is a finite set of vertices or *nodes*. Each node $v_i \in V$ is represented by a tuple ($nodeLabel_i$, $nodeIDList_i$) where
  - $nodeLabel_i$ is a name of the node, composed by
    - $path_i$, the path information starting from the document root and
    - $name_i$, the identifier of v_i which contains the property name
  - $nodeIDList_i$ is a *list* of $nodeID$s, that specifies in which JSON documents a node v_i occurs. A $nodeID$ is constructed by $docID_j : i$ where
    - $docID_j$ is the ID of the JSON document J_j within a Collection C
    - i is a unique node number within the JSON document
- $E \subseteq V \times V$ is a set of directed *edges* or arcs of SG with
  - $e \in E$ is represented by a tuple $(v_k, v_l, IDList_l)$, with v_k, $v_l \in V$, and $v_k \neq v_l$. $IDList_l$ is a list of $nodeID$s. It specifies for a node v_l under which parent v_k it occurs.

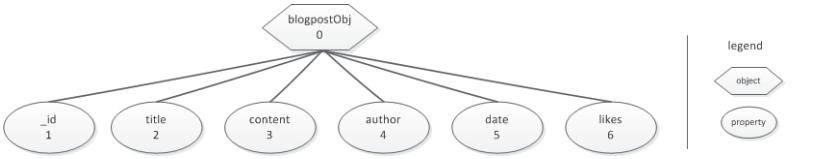
4.3 Example for JSON Schema Extraction

In the following, we provide an example. In a `blogpost` application, we assume the two entities (JSON documents) from Figure 1. The structure representation of the `blogpost` documents is shown in Figure 4(a). We assign unique node numbers in *preorder*. This simple numbering scheme is sufficient here, because the numbers only serve to identify the components during schema extraction. We assume that the JSON documents are not changed during schema extraction.

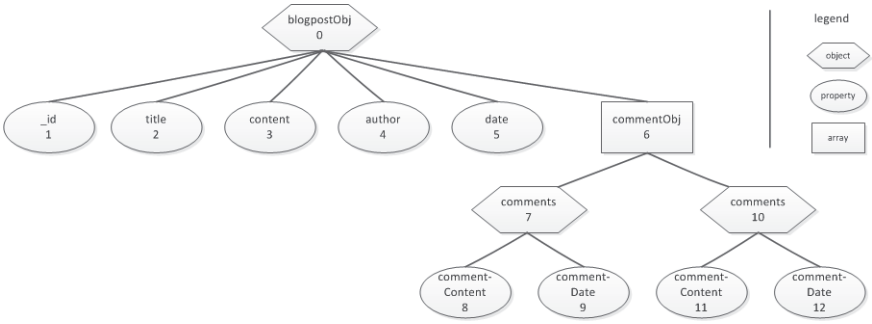
We want to derive the *structure identification graph* SG that represents the structure of all entities. For the JSON documents in Figure 4, we construct the SG in Figure 5. Same structures in the original data are summarized in SG. The node and edge labels reference the nodes from the input data. We next provide detailed algorithm for constructing an SG.

4.4 Schema Extraction Algorithm

We next describe the algorithm for constructing the SG when processing JSON documents. As we have pointed out in Section 2, our approach may be extended to work with



(a) JSON document with key 7



(b) JSON document with key 97

Figure 4: Structure representation of the entities `blogpost` from Figure 1

extensible record stores as well.

Construction of the Structure Identification Graph SG. First, we build the SG from the input data of the NoSQL database. The nodes in the JSON documents are traversed and numbered in preorder. For each node of the input data, an adding or extending of a node in SG `addNode` and an adding or extending of an edge from the node to its parent node `addEdge` is performed. The node numbers specify in which order the SG is constructed.

input data: JSON document collection C
initialize SG: $V = \emptyset$; $E = \emptyset$;
foreach $J_x \in C$ **do**:
  $i = 0$; // initialize counter
  // storing the root node of the JSON document
  SG.addNode(J_x, $\emptyset$, $rootname$, i);
  i=i+1;
  foreach $/path_j/name_j \in J_x$ **do**:
    SG.addNode(J_x, $path_i$, $name_i$, i);
    p= J_x.getID($path_i$);
    SG.addEdge(J_x, $path_i$, $name_i$, p)
    i=i+1;

When a node is added, we distinguish two cases: if the node does not yet exist in the SG, we *add the node*. The node is stored with a node name and with a list that contains only one reference: the `docID` of the current node and the unique node number `i`. If the node already exists in the SG (since the same node occurred in some other input document) then the current node information is appended to the list that is stored with the node.

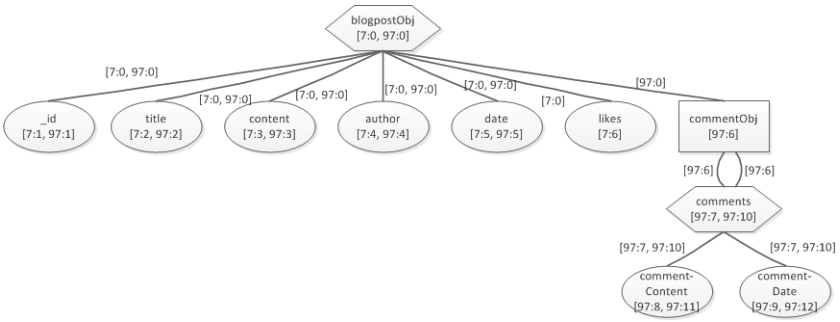


Figure 5: Structure identification graph for the sample entities

def SG.addNode(J_x, $path_i$, $name_i$, i):
  if $(/path_i/name_i) \in SG$:
    $V : (/path_i/name_i, list) \longrightarrow (/path_i/name_i, list$ + J_x.getDocID(): $i)$
  else
    $V := V \cup (/path_i/name_i, [J_x$.getDocID() : $i])$

Adding an edge is similar. If the edge does not yet exists in the SG, it is added, otherwise the edge list is updated by appending the `docID` and the unique node number `p` of the parent node.

def SG.addEdge(J_x, $path_i$, $name_i$, p):
  if $(/path_i/name_i) \in SG$:
    $E : (/path_i, /path_i/name_i, list) \longrightarrow (/path_i, /path_i/name_i, list$ + J_x.getDocID(): $p)$
  else
    $E := E \cup (/path_i/name_i, J_x$.getDocID(): $p])$

The traversal of the JSON documents in *preorder* ensures that the parent nodes are processed before their children.

Schema extraction. In the next step, we derive the JSON schema from the SG.

input data: SG - Structure Identification Graph
foreach $/path_i/node_i \in SG$ **do**:
  generate JSON description for $node_i$;
  if (SG.getNodeList($path_i$).length() == SG.getEdgeList($path_i$, $name_i$).length()):
    $/path_i/node_i$ is required
  else:
    $/path_i/node_i$ is optional

We give an example illustrating required and optional nodes. In the edge lists, only those IDs occur that are also available in the parent node lists. Furthermore, each ID in the

node and edge lists is *unique* (because it is a combination of the documentID and a unique nodeID). So each ID can occur in the node and edge lists at least once. Combining these two facts, we can conclude: if the lists coincide (then the list lengths are identical), the property is required. In Figure 6, the property `title` is *required* because the edge list and the node list of the parent node coincide. If the edge list contain fewer IDs than the parent node list then the property is optional. An example for that case is the property `likes` in Figure 6. It is *optional* because the edge list is a subset of the parent node list and the list length of the edge list is smaller than the list length of the parent node list.

Since the calculation of the list length is more efficient than the comparison of the complete edge and parent node lists, it is used in the algorithm given above.

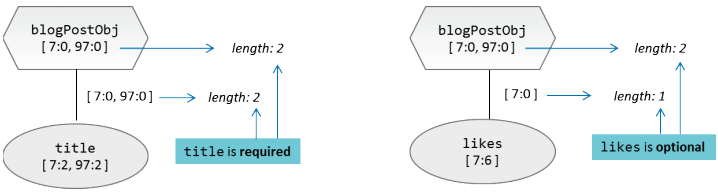


Figure 6: Required and optional properties in the structure identification graph SG

Finding the type of schema components. The above algorithm introduces the general idea of schema extraction from JSON documents. Its implementation is slightly more involved, since it additionally derives the *data type* (string, number, boolean, array, or object) of each schema component from the input JSON documents. This type information is also stored in the nodes of the schema identification graph SG.
If in the input instances the same properties with different data types exist, a union data type, for instance `"oneOf":[{"type":"string"},{"type":"integer"}]` is stored in the JSON schema.

4.5 Detection of Structural Outliers

The previous section described the JSON schema generation from the structure identification graph. The graph with its nodes, edges, node labels, and edge labels delivers even more information than merely a valid schema. We can use it to derive *statistics*, to find *outliers* in the data, and to calculate *measures* that capture how regular the documents within a JSON collection are. Figure 7 visualizes these subtasks within the overall workflow.

Adding statistics to a JSON schema. Adding statistics during JSON schema extraction is a straightforward extension. For each component of the schema, we can record the information how often this component occurred in relationship to its parent node. This information can also be derived from the SG. In our example from Figure 5, the property `likes` occurred in 50 % of the JSON documents. The information can be helpful to

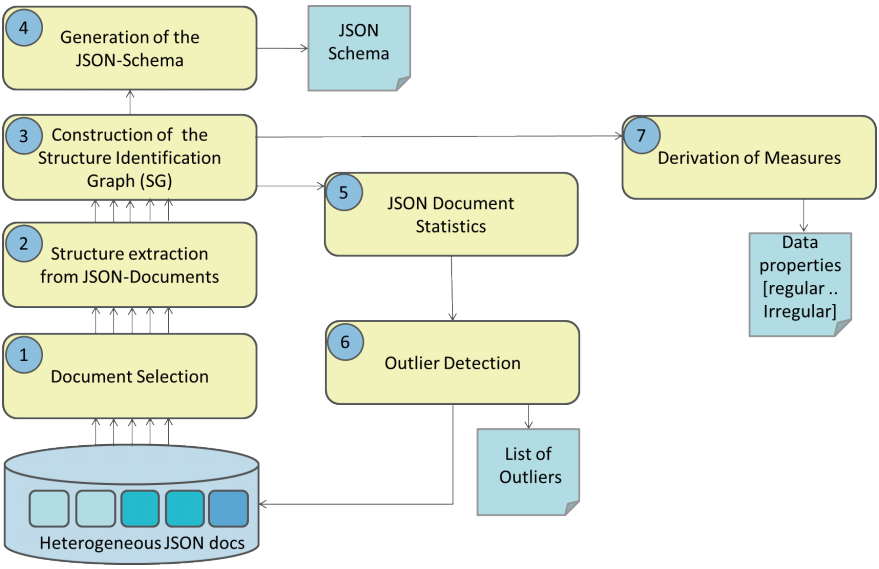


Figure 7: Schema extraction, detection of outliers, and calculation of measures

understand the input data.

Errors and Outliers in NoSQL Databases. Before we start with the outlier detection, we want to motivate why errors or outliers frequently occur in NoSQL databases. The main reason is that most NoSQL database systems do not check any structural constraints, so all well-formed JSON documents can be stored. Often, the data is collected over long periods of time. Different applications and different users have added and changed data. The lack of schema management tools makes it difficult to detect data containing errors or diverging property names.

For this reason, we are looking for structural outliers (that could be errors) with our method. Within our algorithm we cannot distinguish between

- structures that only occur rarely (schema divergences) and
- actual errors in the NoSQL data.

Deciding whether a finding is correct or an error requires user interaction. We automatically suggest *candidates*, but the *final error classification* remains a task for the user.

Detecting outliers in NoSQL data. With the structure identification graph SG, we can find all structural outliers, i.e., all variants that are under a given threshold ϵ. It delivers two kinds of outliers:

1. *Additional properties.* We can derive which properties exist only in some JSON documents (in less than ϵ (in %) of the documents). The list of these properties is

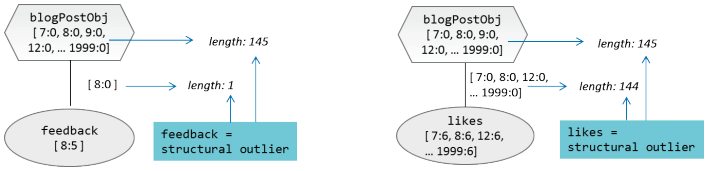
determined by calculating for each $v \in V$:

$edgeList$= SG.getEdgeList($/path_i$, $/path_i/name_i$)
$parentList$ = SG.getNodeList($/path_i$)

If $\frac{|edgeList|}{|parentList|} * 100 < \epsilon$ then a node only occurs rarely. JSON documents that contain this node are outliers. $outliers$ = $edgeList$ delivers references to all JSON documents that contain the node. Figure 8(a) shows an example.

2. *Missing properties.* Another kind of structural outliers that we can find with the SG are missing properties. If a property occurs in nearly all data sets and is only missing in some JSON documents ($\frac{|edgeList|}{|parentList|} * 100 > 100 - \epsilon$)
then we obtain the list of outliers as $outliers = parentList - edgeList$. Figure 8(b) shows an example.

In the running example consisting only of 2 documents, we cannot detect outliers, because the example does not contain enough documents. We therefore extend the sample structure identification graph SG and represent both cases in Figure 8.



(a) Additional components (b) Missing components

Figure 8: Finding outliers in the structure identification graph SG

With a semiautomatic approach, the user can decide whether the structure is correct and whether it shall be considered in the schema representation. Otherwise, the user decides that a property shall not considered in the target schema and classifies the corresponding JSON documents as errors. We do not need to touch the stored NoSQL database again, the information which data sets are incorrect can be derived from the SG.

In summary, our approach does not support automatic cleaning and correction of the data, to avoid information loss.

4.6 Measures for Variability

In this section, we introduce and calculate measures for the *degree of coverage* of the JSON documents, that capture the *structural homogeneity* of a document collection.

Let's start with some definitions. Each property in a JSON document J_i can be addressed by a complete path expression, starting from the root. A property $p_k = /path_k/name_k \in J_i$ if the property with the specified `path` and `name` is available in J_i.

The *document size* $|J_i|$ is defined as the total number of properties in a JSON document.

The common properties of two JSON documents J_i and J_j are defined as follows:

$$J_i \cap J_j = \{p_k \mid p_k \in J_i, p_k \in J_j\} \tag{1}$$

Degree of coverage. We further define the *degree of coverage* between two JSON documents J_i and J_j.

$$cov(J_i, J_j) = \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_i|} + \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_j|} \tag{2}$$

The coverage specifies the overlap of two documents. It delivers a value between 0 and 1 that captures the structural similarity of two documents. We can generalize the coverage definition for an arbitrary subset $J_i..J_k$ of k JSON documents:

$$cov(J_1..J_k) = \frac{1}{k} \cdot \sum_{i=1}^{k} \frac{|\bigcap_{j=1}^{k} J_j|}{|J_i|} \tag{3}$$

This measure considers the coverage of the documents $J_1..J_k$ of a collection C. It can efficiently be calculated. If the cov value is 0 the JSON documents are completely heterogeneous, if the value is 1 then all documents have exactly the same structure. A disadvantage is that this measure is strongly influenced by a few JSON documents, even one JSON document with divergent structure changes the result. Other measures, for instance the Jaccard coefficient often used for determining the similiarity has the same disadvantage.

$$jac(J_1..J_k) = \frac{\bigcap_{j=1}^{k} J_j}{\bigcup_{j=1}^{k} J_j} \tag{4}$$

One JSON document with divergent structure can cause that the Jaccard measure delivers the value 0.

As an alternative, it is possible to introduce a more sophisticated measure that considers all subsets S of a JSON collection C= $J_1..J_n$ with at least 2 documents. With $|S|$ the number of documents in a subset S and $|M|$ the size of M, we can now calculate:

$$M = \{S|S \subseteq (J_1..J_n), |S| >= 2\} \tag{5}$$

$$cov_{sub}(J_1...J_n) = \frac{1}{|M|} \cdot \sum_{S \in M} cov(S) \tag{6}$$

This measure delivers better results especially for collections that are homogeneous and contain only one or a few JSON documents that feature another structure. An disadvantage is that this measure cannot be efficiently calculated for large collections.

That's why we proceed with the simpler coverage measure (Formula 3), and define on it the *supplement* of each JSON document J_i within a JSON collection $C = J_1..J_n$.

$$sup(J_i, C) = \frac{|J_i| - |\bigcap_{j=1}^{n} J_j|}{|J_i|} \tag{7}$$

Required and optional schema components. We consider properties that occur in all JSON documents $J_1..J_n$ of a collection C required properties. Properties are optional if they occur in at least one, but not in all JSON documents of a collection C.

$$req(C) = \{p \mid \forall J_i \in C : p \in J_i\} \tag{8}$$

$$opt(C) = \{p \mid \exists J_i, J_j \in C : p \in J_i \wedge p \notin J_j\} \tag{9}$$

In JSON documents with a homogeneous structure, most properties are required. Heterogeneous JSON documents mainly have optional properties. These characteristics are also reflected in the schemas of the JSON document collections.

We want to determine the degree of coverage with Formula (3) for our running example from Figure 1.

$$cov(J_i, J_j) = \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_i|} + \frac{1}{2} \cdot \frac{|J_i \cap J_j|}{|J_j|} = \frac{1}{2} \cdot \frac{5}{6} + \frac{1}{2} \cdot \frac{5}{12} = 0.625 \tag{10}$$

The degree of coverage can be calculated in parallel to the schema extraction. $\bigcap_{j=1}^{n} J_j$ can be derived by traversing SG. The counting of the properties of each JSON document $|J_i|$ and storage of this number is an easy-to-realize extension of the algorithm.

5 Reduced Structure Identification Graph (RG)

We also have implemented a second variant of the schema extraction approach that uses a **R**educed Structure Identification **G**raph (RG). We have seen in Section 4 that the schema extraction algorithm does only base on the node and edge lists lengths (not on the list elements). In this variant, the graph does not contain references to the original JSON

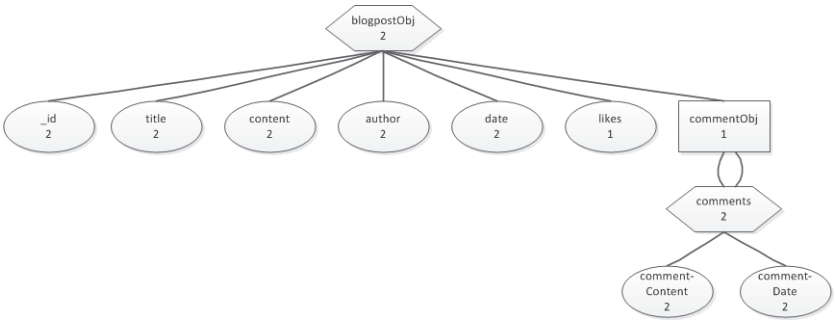


Figure 9: Reduced structure identification graph (RG) for the blogpost application

documents, we only count the *number of occurrences* and store it for each *node* of the graph. Figure 9 shows RG for the running example.

We can introduce this reduced structure (in contrast to the original approach [MLN00]) because JSON documents are more simple in structure than XML documents. Let's take a small XML example for illustrating it (figure 10):

```
<book>
    <author> .. </author>
    <author> .. </author>
    <title> .. </title>
    ...
</book>
```

```
<book>
    <editor> .. </editor>
    <title> .. </title>
    ...
</book>
```

Figure 10: Sample XML documents

If we would count the occurring elements in both XML documents then we would get the result that `book`, `title` and `author` occur twice. The elements `book` and `title` are required in our small example, but `author` is optional, because the element is not available in the second XML-document. So, in XML the number of occurrences is not sufficient to find out the required and optional elements. In contrast, in JSON documents, each property (with its complete path) is unique in the JSON documents. The number of occurrences corresponds with the number of JSON documents containing the specified property.

That's why, we can use the simpler graph structure RG for the JSON schema extraction algorithm. With this data structure, we can derive the schema, we also can determine *statistics* and we can derive the *coverage value*. It is not possible to determine *outliers* with the data structure. The table in Figure 11 compares both alternatives of the schema extraction algorithm.

(*) Also in this variant, it is possible to detect outliers. From the JSON schema and additional statistics, we can derive which properties p_i occur only rarely ($< \epsilon$). In the next step, we query the original JSON data set and find out the documents that contain the property p_i. In MongoDB, for example, we can execute the following query:

	Structure Identification Graph (SG)	Reduced Structure Identification Graph (RG)
JSON Schema Extraction	+	+
Document Statistics	+	+
Similarity Measures	+	+
Outlier Detection	+	- (*)

Figure 11: Comparison of the structure extraction and outlier finding with a structure identification graph SG and a reduced structure identification graph RG

```
db.collection.find({pi: {$exists: true}})
```

If we find out that a property p_j occurs in the majority of documents ($> (100\% - \epsilon)$), we select the matching JSON documents from the NoSQL database with the following query:

```
db.collection.find({pj: {$exists: false}})
```

We only have to consider that at the moment not all NoSQL databases support negations in queries. For instance, in the current versions of Cassandra we cannot express queries containing a `not exists`.

6 Experiments

We have implemented our schema extraction algorithm and conducted experiments on the configuration database of the Wendelstein 7-X project [SLB+12]. This database has been in use for several years to record configurations of plasma experiments. It currently holds more than 120 different collections to record information on devices, as well as parameters that control the settings of experiments. Most collections within this database are structured and regular, yet there are also structural outliers. Until today, the database has been used without any explicit schema description.

Our implementation has extracted one JSON schema for each collection. We implemented the two alternatives for schema extraction as introduced in Sections 4 and 5 in Python. Our experiments ran on an Intel Core i7-461000U CPU @ 2.70GHz machine with 8.00 GB RAM. In most applications scenarios, schema extraction is run as a batch process, so we do not require runtimes that allow for real time user interactions. However, runtime is still a critical criterion for the feasibility of schema extraction. Thus we study the runtime of our algorithms to acquire ballpark numbers, and to determine whether the method is even applicable in practice. Moreover, we decided to use an off-the-shelf commodity machine, rather than a powerful data server, since we want to observe if the algorithm can be executed without special hardware.

The following performance measurements show the execution times of schema extraction using a structure identification graph introduced in Section 4. We represent the performance measures for the 30 most data-intensive Collections of the Wendelstein-7-X dataset.

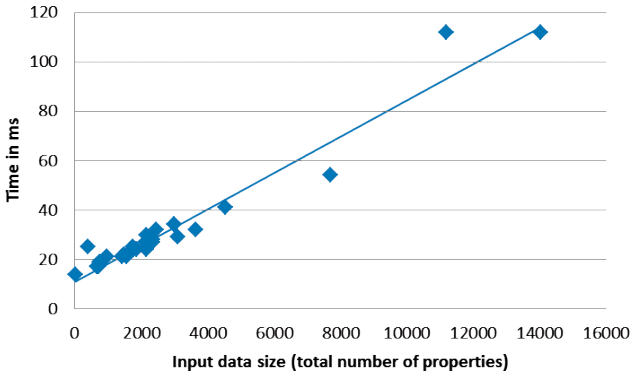


Figure 12: Schema extraction for the different collections of the Wendelstein data

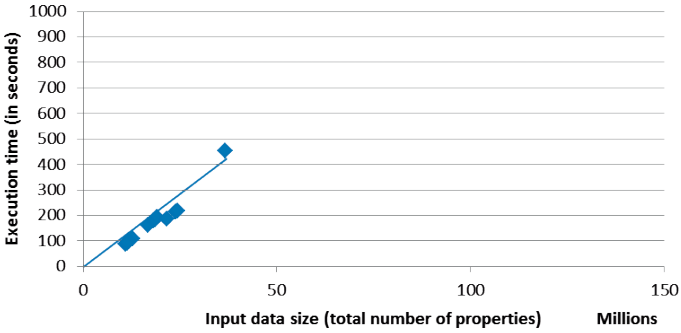
On the x-axis, the size of the input data is shown. We determine the total number of structural information that means the sum of the number of properties, arrays, and objects in the JSON collection. As an example: One collection contains 206 JSON documents, each JSON document has 22 properties, so the input data size of this collection is 4532. Our experiment shows linear growth with the size of the input data (see Figure 12).

The schema extraction approach based on the Structure Identification Graph SG (Section 4) works very well for small data sets. An obvious bottleneck is that all nodes and edges carry references to the original data. For collections containing many data sets these lists become very large. For that, we applied the alternative approach based on the RG that was introduced in Section 5.

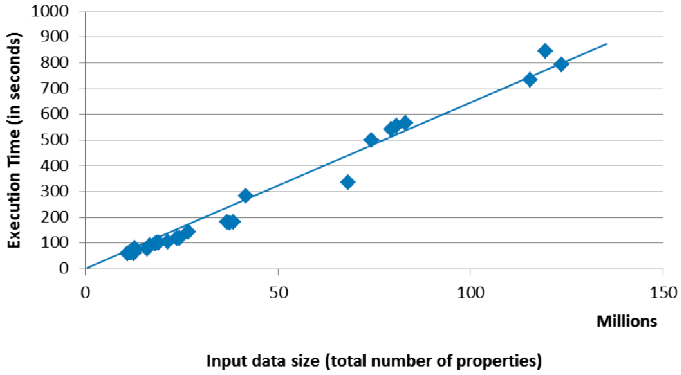
To compare both approaches, we tested them with yet another dataset. We chose a larger MongoDB database storing about 800 collections in 150 GigaByte of data. The database contains statistical data of Web Performance Measures. It describes different technical system parameters, logdata, and different performance data of several Web Information Systems of the Pagebeat project [FBH+14]. Figure 13 shows some results of the schema extraction for this data. The SG construction for the larger data set 13(a) is only possible for data sets up to 40 million input nodes (about 500 Mega Byte when stored in MongoDB). The method with RG could be realized for all collections of the data set up. Again, the algorithm runtime is linear in the size of the input data.

7 Related Work

There is a large body of work on schema extraction from XML documents ([MLN00], [DCjWS06], [HNW06]). Our approach builds upon the work of Moh, Lim, and Ng, who have developed a DTD Miner for extracting a schema for a given set of XML documents [MLN00]. They capture the structure of each XML document with a tree, and then



(a) Schema Extraction with SG



(b) Schema Extraction with RG

Figure 13: Schema extraction on different MongoDB collections on Web usage data

derive a graph representation which summarizes all structural variants. Finally, a DTD is derived that identifies optional elements and repeating elements. In deriving a JSON schema, as done in this paper, the document order and repetition of elements do not play a role. This makes it possible to build the reduced graph, as discussed in Section 5.

The authors in [BNSV10] give a comprehensive survey over the extraction of DTDs and regular expressions from XML documents in particular. They point out the challenges imposed by the order of siblings in XML documents, and the fact that cross-references between nodes actually define graphs, rather than trees. With JSON documents, however, the order among sibling nodes does not play a role.

In [BCNS13], the authors propose a type system for JSON objects as well as a type inference algorithm. This serves as a basis for defining the semantics of JSON query languages.

The JSON type system in [CGSB12] comes with a MapReduce-based type inference al-

gorithm. In the context of NoSQL data stores, scalability is vital, since we may need to handle data sets in the *big data* category. At the time of writing this paper, there are no benchmarks effectively demonstrating the scalability of this approach yet.

In data exchange, schema extraction plays a vital rule. IBM's BlueMix portfolio [IBM14] offers a *schema discovery* component for importing JSON data from Cloudant into the data warehousing product Dynamite. Any received data that does not match this schema is sent to a bucket, where it can then be inspected manually.

In [KSS14], we introduce the idea of an external schema management component for NoSQL databases. In [SKS13] we further propose a language for declaring schema evolution operations for extensible record stores and for NoSQL document stores.

8 Summary and Future Work

In this article, we introduce two variants of schema extraction for NoSQL databases. The first one is similar to the DTD Miner [MLN00] that was developed for XML documents. The second algorithm relies on an optimization that we can make because JSON documents are simpler in structure than XML documents. With both approaches, we can generate a JSON schema from an available data set. We show that the internal data structure that we use for schema discovery can also be used for collecting statistics on the input data, as well as for detecting structural outliers. For instance, we can derive a similarity measure that captures how regularly the data within a collection is structured.

We tested our approaches on different JSON collections managed in MongoDB. With the Web Statistic data from the Pagebeat project [FBH+14], our outlier detection found several outlier documents that were not previously known to the owners of this data. Knowing about outliers is an important prerequisite for improving the data quality.

In our long term vision, we see our schema extraction algorithm as part of a powerful schema management component for NoSQL databases. Systems that do not actively manage the schema actually force software developers to maintain the schema constraints within in the application logic. To assist developers in their daily work, we are designing dedicated tools for handling and managing the schema for NoSQL data stores, while paying full respect to their strongpoint, their schema-flexibility.

References

[Apa13] Apache Cassandra, 2013. `http://cassandra.apache.org/`.

[BCNS13] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and Dynamic Semantics of NoSQL Languages. In *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 101–114, New York, NY, USA, 2013. ACM.

[BLS99] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes - A Survey*. SIAM Monographs of Discrete Mathematics and Applications, 1999.

[BNSV10] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2), 2010.

[Cat10] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[CGSB12] Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Universität Della Basilicata. Typing Massive JSON Datasets, 2012.

[DCjWS06] Theodore Dalamagas, Tao Cheng, Klaas jan Winkel, and Timos Sellis. A methodology for clustering XML documents by structure. *Information Systems*, 31:187–228, 2006.

[Ecm13] Ecma International. *The JSON Data Interchange Format*, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[FBH+14] Andreas Finger, Ilvio Bruder, Andreas Heuer, Martin Klemkow, and Steffen Konerow. PageBeat - Zeitreihenanalyse und Datenbanken. In *GI-Workshop: Grundlagen von Datenbanken'*, 2014.

[HNW06] Jan Hegewald, Felix Naumann, and Melanie Weis. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *ICDE Workshops*, page 81, 2006.

[IBM14] IBM. IBM Blue Mix, 10 2014. `https://ace.ng.bluemix.net/`.

[Jav03] Java Data Objects Expert Group. *JSR 12: Java Data Objects*, 2003.

[Jav09] Java Persistence 2.0 Expert Group. *JSR 317: Java Persistence 2.0*, 2009.

[JSO14] JSON Schema Community. *JSON Schema*, 2014. `http://json-schema.org`.

[KSS14] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Datenbanken ohne Schema? - Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schema-flexiblen NoSQL-Datenbanksystemen. *Datenbank-Spektrum*, 14(2):119–129, 2014.

[MLN00] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keon Ng. DTD-Miner, A Tool for Mining DTD from XML Documents. In *Proc. WECWIS*, 2000.

[SBT+12] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, et al. F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[SHKS15] Uta Störl, Thomas Hauff, Meike Klettke, and Stefanie Scherzinger. Schemaless NoSQL Data Stores Object-NoSQL Mappers to the Rescue? In *Proc. 16. BTW, March, 2015 Hamburg, Germany*, 2015.

[SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. *Proc. DBPL*, CoRR, abs/1308.0514, abs/1308.0514, 2013.

[SLB+12] Anett Spring, Marc Lewerentz, Torsten Bluhm, Peter Heimann, Christine Hennig, Georg Köhner, Hugo Kroiss, Johannes G. Krom, Heike Laqua, Josef Maier, Heike Riemann, Jörg Schacht, Andreas Werner, and Manfred Zilker. A W7-X experiment program editor - A usage driven development. In *Proc. 8th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research*, pages 1954 – 1957, 2012.

[Tiw11] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.

Scaling Out the Discovery of Inclusion Dependencies

Sebastian Kruse, Thorsten Papenbrock, Felix Naumann

Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam
firstname.lastname@hpi.de

Abstract: Inclusion dependencies are among the most important database dependencies. In addition to their most prominent application – foreign key discovery – inclusion dependencies are an important input to data integration, query optimization, and schema redesign. With their discovery being a recurring data profiling task, previous research has proposed different algorithms to discover all inclusion dependencies within a given dataset. However, none of the proposed algorithms is designed to scale out, i.e., none can be distributed across multiple nodes in a computer cluster to increase the performance. So on large datasets with many inclusion dependencies, these algorithms can take days to complete, even on high-performance computers.

We introduce SINDY, an algorithm that efficiently discovers all unary inclusion dependencies of a given relational dataset in a distributed fashion and that is not tied to main memory requirements. We give a practical implementation of SINDY that builds upon the map-reduce-style framework Stratosphere and conduct several experiments showing that SINDY can process huge datasets by several factors faster than its competitors while scaling with the number of cluster nodes.

1 Discovering Inclusion Dependencies

Given a relational dataset, an inclusion dependency (or short IND) is a constraint expressing that all values of some column (combination) A are also found in some other column (combination) B. This is formally expressed as $A \subseteq B$ and read as "A is included in B". In the context of this IND, A is called the *dependent* column and B the *referenced* column.

This rather simple type of dependency has a broad range of applications, the most prominent one being foreign key discovery. In a dataset that consists of multiple relations, foreign keys describe how their records relate to each other, thus, their knowledge is crucial to any further work with this dataset. However, for various reasons this knowledge can be lacking: datasets are obtained as dumps in some export format that does not capture the schema of the dataset but merely its content; not all databases are capable of enforcing foreign key constraints; and even if so, capturing these constraints is sometimes avoided for performance reasons as the foreign key enforcement involves some computational overhead. Besides foreign keys, INDs are valuable knowledge to many other applications [CTF88, Gry98, LV00], like schema redesign (e.g., detect duplicate information) and data integration (e.g., joining tables from multiple data sources).

As opposed to the detection of other data dependencies, such as functional dependen-

cies [HKPT99, ASN14] or uniqueness constraints [SBHR06, HQRA+13], the IND detection problem suffers the general deficiency of not being able to prune candidates early on: If one wants to know whether $A \subseteq B$ holds for some columns A and B, one usually has to read both columns completely: To confirm this dependency, *each value of* A must be shown to be in B, and disproving it requires to show for a value of A that *each value of* B is different to this value. To this end, previous IND discovery algorithms mostly comprise a *data reorganization phase* to make columns better comparable, which takes the major share of the runtime. Pruning techniques are only applied in the subsequent *comparison phase*, where it does not pay off much.

Given this problem, a promising and new approach to push the envelope of efficient detection is to scale out. We propose SINDY (scalable inclusion dependency discovery), an algorithm that distributes the discovery of INDs among multiple nodes in a computing cluster. In particular, the usually expensive, I/O-bound data reorganization benefits from high I/O throughput rates that can be achieved thereby.

As SINDY reuses some ideas and concepts of previous algorithms, we first present related work in Sec. 2. Then, in Sec. 3, we explain the distributed setting of our approach and explain the discovery of INDs in distributed computing environments in detail in Sec. 4. We conducted various experiments with SINDY on mostly real-world datasets, the results of which we present in Sec. 5. Eventually, we conclude in Sec. 6.

2 Related Work

The discovery of all inclusion dependencies is an important and well studied profiling task [CFP82]. A good part of the previous work deals with the more specific problem of foreign key detection [RAB+09, ZHO+10]. However, since in general only a few INDs correspond to actual foreign keys, these approaches do not aim at finding all the INDs of a dataset and rather optimize for precision and recall than for runtime. In consequence, they are inapplicable to other IND use cases, such as query optimization [Gry98], integrity checking [CTF88], or schema matching [LV00] that require complete results.

Bell and Brockhausen devised a system that starts by collecting data statistics from which it derives a set of IND candidates [BB95]. These IND candidates are then successively checked against the dataset using SQL `join`-statements. This checking procedure also uses intermediate results to prune yet unchecked candidates. Despite the pruning, the proposed algorithm is much slower than more recent works in this area, mostly due to the exhaustive use of SQL `join`-statements for the IND validations.

De Marchi et al. identified the validation of INDs as the bottleneck of the discovery process and developed an algorithm that greatly optimizes it [DLP09]. The algorithm consists of two phases: In the first phase, it transforms the database into an inverted index pointing each value to the set of attributes containing the value. In the second phase, the algorithm then validates IND candidates against the inverted index by intersecting their referenced attribute groups with all attribute sets of the inverted index that also contain the dependent attribute group. The set intersections are an efficient way to validate many IND candidates

simultaneously without querying the dataset for each check. Related works still outperform De Marchi's algorithm, because it needs multiple passes over the inverted index and waives any pruning [BLNT07]. In our case, where the computation is distributed among computers, we can perform only minor pruning, but the heavy use of parallelization compensates this disadvantage. The need to pre-calculate the whole inverted index also makes the De Marchi algorithm inapplicable for datasets that do not fit into main memory. Our distributed algorithm, which targets in particular such large datasets, solves this issue with data distribution and dynamic memory handling.

Bauckmann et al. presented the SPIDER algorithm [BLN06], which builds upon an adapted sort-merge-join. Like the join-algorithm, SPIDER executes in two phases: In the first phase, it sorts the values of each attribute, removes duplicate values, and writes the resulting sorted lists to disk. In the second phase, SPIDER then simultaneously iterates the sorted lists and can thereby determine for each value the set of attributes that contain this value. Like De Marchi's algorithm, SPIDER intersects these attribute groups with all IND candidates whose dependent attributes occur in the attribute group in order to exclude invalid IND candidates. The dynamic construction of attribute groups from the sorted value lists allows SPIDER to omit all those attributes from the validation procedure that do not appear in any valid IND candidate any more. Although this pruning makes SPIDER an efficient IND discovery algorithm on single compute nodes, it prevents the algorithm from being efficiently parallelized. SPIDER also requires an open file for each attribute; this is expensive and limited by many operating systems. Our algorithm, in contrast, is not subject to such restrictions.

3 Distributed Setting

Consider the example database depicted in Tab. 1 with discographic data over two relations. Its six columns yield $6 \cdot (6-1) = 30$ IND candidates, one for each column pair.[1] A closer look reveals that this database comprises six INDs, e.g., $album \subseteq track\_alb$, $position \subseteq track\_pos$, and $album \subseteq title$. To extract these six INDs from the set of IND candidates, SINDY applies a special procedure that requires only a single pass over the data rather than comparing each candidate separately. In particular, this procedure can be scaled out over many computers as we detail in Sec. 4.

Hence, SINDY is designed to run on clusters. In such a distributed scenario, multiple independent computers (*workers*) are available for the execution of the algorithm. These workers have no shared state or control, which renders the design of distributed algorithms particularly challenging. However, the workers are interconnected via a network and can exchange messages and data. Furthermore, the input datasets are (redundantly) distributed over these nodes, e.g., as CSV files in an HDFS[2]. Facing these circumstances, our main goal is to distribute the incurring workload among the workers in a manner that utilizes all their available resources like CPU cycles, main memory, and especially I/O bandwidth

[1] In practical scenarios, this number can be sometimes reduced by incorporating already known metadata such as datatypes [DLP09] or minimum and maximum values.

[2] `http://hadoop.apache.org/`

album	position	title
Thriller	4	Thriller
Thriller	5	Beat it
Back in Black	1	Hells Bells
Back in Black	6	Back in Black

(a) Example table *Tracks*.

track_alb	track_pos	name
Thriller	4	Michael Jackson
Thriller	4	Vincent Price
Thriller	5	Michael Jackson
Thriller	5	Eddie van Halen
Back in Black	1	AC/DC
Back in Black	6	AC/DC

(b) Example table *Artists*.

Table 1: An example database with two relations.

simultaneously. Hereby, we want to achieve increased performance as more workers – and thus more resources – are available for the computation.

Map/Reduce frameworks like Apache Hadoop that support this goal have gained remarkable attention over the last years and are nowadays widely used. By means of a simple programming model, they allow to create distributed applications, that can be executed in a fault-tolerant and scalable manner (wrt. available main memory as well as the number of workers). Following this trend, generalizations of the map-reduce paradigm have been researched, especially by the Stratosphere research project. Amongst others, in Stratosphere complex processing pipelines replace the static map-reduce structure [ABE+14].

We particularly took care that SINDY can be easily implemented within such frameworks, to benefit from their aforementioned advantages. However, using such frameworks comes with the drawback that they mostly offer high-level interfaces that only give control over the data processing to a certain level of detail. This is similar to formulating a SQL query, where one also cannot specify the algorithms to be used for joins, sortations etc. Nevertheless, we believe that this trade-off is worthwhile for IND discovery, because frameworks like Stratosphere can leverage this abstract specification for optimization purposes and are capable of intelligently picking suitable data processing strategies.

4 Distributing the discovery of INDs

The goal of IND discovery is to state for every irreflexive column pair $(A,\ B)$ in the given relational dataset if $A \subseteq B$ holds. However, instead of checking each column pair individually, SINDY efficiently checks all pairs in a single pass over the data. We provide the foundation for our approach with an alternative formalization of INDs: Let $O = A \bowtie B \bowtie \ldots$ be the full outer join of all columns in a database, then

$$A \subseteq B \Leftrightarrow \forall t \in O : (t[A] \neq \bot \Rightarrow t[B] \neq \bot) \tag{1}$$

Existing algorithms already use this formalization implicitly. For instance, SPIDER [BLN06] performs an optimized sort-merge join to find INDs and MIND [DLP09] pivots the database for this reason. Then, they iterate the join result once or more often to find the inclusion dependencies as described above. SINDY consists of a join and a checking phase, too, but

performs both phases in a distributed manner. In the following, we explain these phases in detail and afterwards describe how they are implemented in Stratosphere.

4.1 Join columns to attribute sets

In the first phase, the full outer join $O = A \rtimes B \rtimes \dots$ of all columns in the database is computed. Since each tuple in the join product is of the form $t_{O,v} = (v_A, v_B, \dots)$ with $v_X \in \{v, \perp\}$ (i.e., each tuple can only contain null values and one distinct value v), these tuples can be simply represented as $r(t_{O,v}) = (v, \{X | t_{O,v} \neq \perp\})$. Additionally, Equation 1 neither makes use of the actual values in O nor profits from tuples where the exact same attributes are null; therefore, we can discard the value and opportunely remove duplicates. As the result of this process, SINDY yields *attribute sets* that represent O.

Figure 1 exemplifies the dataflow of the outer join procedure on two workers with some tuples of the example from Sec. 3. To find the attribute sets, each worker initially reads its local share of the input data and splits each read record into *cells*, i.e., tuples with a value and a singleton set containing the corresponding attribute. For instance, the record ("Thriller", 4, "Thriller") from the example table $tracks$ is split into the three cells ("Thriller", {album}) and (4, {position}), and ("Thriller", {title}). SINDY now reorders all cells among the workers of the cluster: For n workers with IDs $0, \dots, n-1$ we define the target worker for each cell from its value by means of a partition function $p\colon \mathbb{V} \to \{0, 1, \dots, n-1\}$. It is thereby guaranteed that cells with the same value are placed on the same worker. Ideally, the partition function distributes the values evenly among the workers. A suitable choice for p is $p(v) \stackrel{def}{=} \mathit{hash}(v) \bmod n$.

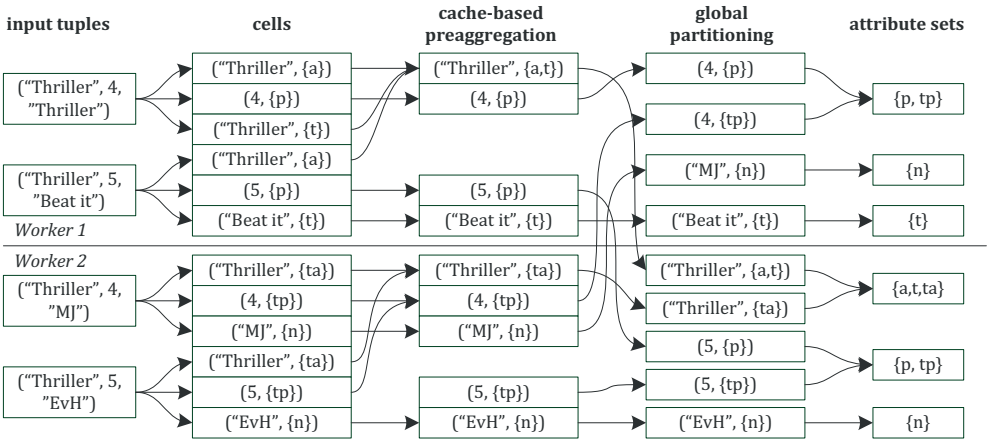


Figure 1: Dataflow for a distributed full outer join on all columns of a dataset. The attribute names and some values are abbreviated for the purpose of lucidity.

Finally, each worker groups all its cells by their values and aggregates their attribute sets

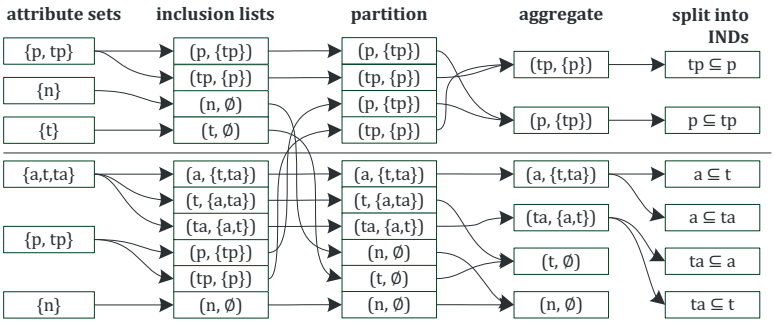


Figure 2: Dataflow for deriving INDs from the outer join product.

using the union operator. The value of each aggregated cell can be discarded, leaving only the attribute sets. Note that the aggregation with the union operator is a commutative, associative operation – it is therefore possible to pre-aggregate the cells on each worker before they are sent to the target worker to reduce network load. However, experiments showed that this almost doubles the runtime. The rationale behind this is that the aggregation for cells requires disk-based execution, which dominates the runtime. Performing an additional pre-aggregation potentially requires each cell to be written and read twice as many times as when performing only the aggregation. Nevertheless, oftentimes columns of a dataset contain only a few distinct values that are then frequently read. It makes good sense to cache cells with values on the reading worker that occur repeatedly and send only cells to the target worker that are not found in this cache yet.

4.2 Derive INDs from the join product

The join product from the previous phase is present as a distributed set of attribute sets. To identify INDs from this product we reformulate the criterion from Equation 1 as it is similarly applied in [DLP09] and [BLN06]:

$$A \subseteq B \quad \Leftrightarrow \quad \forall \text{ attribute set } S\colon (A \in S \Rightarrow B \in S) \quad \Leftrightarrow \quad B \in \bigcap_{S\colon A \in S} S \qquad (2)$$

SINDY performs this check by creating *inclusion lists* for each attribute set as depicted in Fig. 2 for our example. Let S be an attribute set with n attributes, then n inclusion lists of the form $(A, S \setminus A)$ with $A \in S$ are created. These inclusion lists are globally grouped by the first attribute and the attribute sets are intersected. Similar to the union aggregation in the previous phase, the intersection aggregation is associative and commutative and thus allows pre-aggregation as well; the same mechanisms can be reused for this aggregation. Furthermore, the number of attributes in a dataset is usually much lower than the number of values. Therefore, cache-based pre-aggregation seems to be especially suitable here; and once an inclusion list associates an attribute with the empty set, any further aggregation

with an inclusion list for the same attribute is moot and can be omitted. Nevertheless, this approach also allows to handle huge schemata, as we can perform the aggregation of the inclusion lists with disk-based execution if necessary. Finally, the aggregated inclusion lists can be disassembled into INDs: For every inclusion list (A, S), each $A \subseteq B$ $(B \in S)$ is a valid IND, because of Equation 2. Hereby, every inclusion list yields a set of unique INDs, as the attribute A is distinct in every inclusion list.

The logic of SINDY can be directly expressed in terms of map and reduce operators and is therefore well-suited for the implementation on frameworks like Stratosphere. In Fig. 3, we show the execution plan for the unary IND detection. Note that this exact plan can also be implemented with Spark – and in addition, it can be split into two Map/Reduce jobs to be deployed on Hadoop.

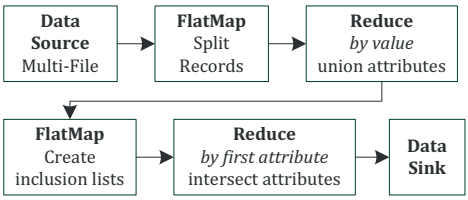


Figure 3: Stratosphere plan for unary IND detection.

5 Evaluation

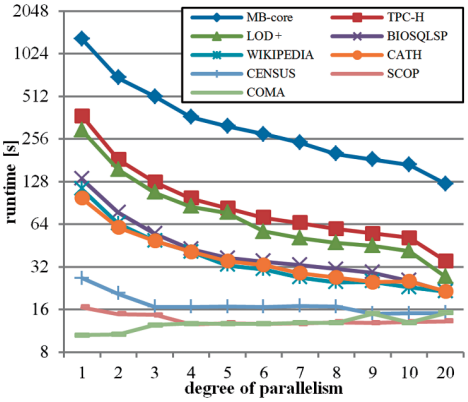
To evaluate SINDY, we performed several experiments with different datasets[3]. Most importantly, we tested its scale-out behavior and compared the runtimes with the state-of-the-art algorithm SPIDER. For our experiments, we used a cluster with a dedicated master node (2.67 GHz Intel Xeon processor and 8 GiB of RAM) and 10 workers with 2.6 GHz Intel Core 2 Duo processor and 8 GiB of RAM of which we dedicated 6 GiB for Stratosphere. All computers were interconnected with a switch router using Gigabit ethernet. We used Hadoop Filesystem 2.2 and Flink 0.6.2[4], which is the most recent version of the Stratosphere software. SPIDER was executed on a machine with two 8-core 2 GHz Intel Xeon processors, 128 GiB of RAM, and a RAID-1 storage.

5.1 Scale-out and comparison

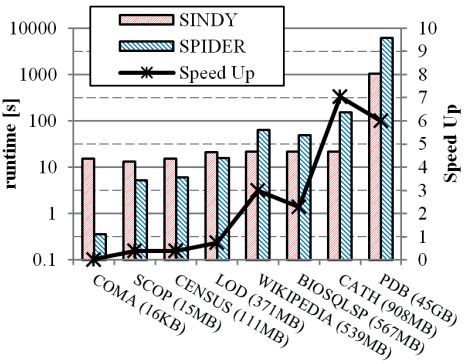
We measured the scale-out behavior of SINDY with small (starting from 16 KiB, COMA) and large datasets (up to 5.8 GiB, MB-core) and present the results in Fig. 4a. We used one to ten workers for the execution and finally added intra-worker parallelization of factor 2 (last measurement). For small datasets that have a short runtime of only a few seconds, the scale-out is not appropriate and can even slow down the execution. For large datasets, though, scaling-out pays off. Adding a second worker can already halve the execution time. For all ten workers, the scale-out is not completely linear, though. We suspect that this is the typical load balancing issue: oftentimes we could see that some workers took

[3] http://hpi.de/naumann/projects/repeatability.html
[4] https://flink.incubator.apache.org/

longer than others to complete their work share. The runtime is determined by the slowest worker. The more workers are involved, the higher is the chance that one of these workers has to cope with an higher-than-average workload.



(a) Scale-out of SINDY on different datasets.



(b) Performance comparison of SINDY and SPIDER.

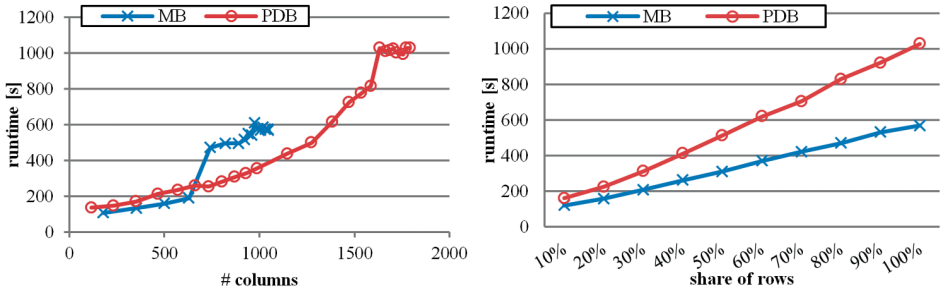
Figure 4: Scale-out behavior and comparison with state-of-the-art.

Next, we compared the runtimes of SINDY with full scale-out to the runtimes of SPIDER (Fig. 4b). As seen before, SINDY does not adapt well to small datasets and is outperformed by SPIDER there. However, for the larger datasets, starting with WIKIPEDIA, the advantages of the scale-out take effect. For our setup with ten workers, the full scale-out potential is apparently reached at around 900 MB. On the largest dataset, SINDY's runtime of 17 minutes in fact saves 85 minutes in comparison to SPIDER. This speedup can be presumably further improved by adding new workers to the cluster, while for the I/O-bound SPIDER the hardware improvements are limited and costly; one could switch to SSDs or extend the RAM to fit datasets in main memory.

5.2 Row and column scalability

Furthermore, we investigated how SINDY behaves as its input data grows. Relational data can grow in two dimensions, namely the number of columns and the number of rows. To obtain meaningful measurements, we based our experiments on the two larger real-world datasets PDB (44.9 GiB) and MB (26.8 GiB).

Figure 5a shows the column-scale behavior of SINDY. To vary the number of columns for both datasets, the original input files remained unchanged, but for each test execution, we used only the first k columns of each file. The idea behind this is to let the amount of processed data grow approximately proportionally with the number of columns on the face of an uneven distribution of the data volume among the input files. For the MB dataset, the processing time increases irregularly with the number of processed columns. In particular, there is a large hop at the area of around 750 columns. We explain this jump, because



(a) Behavior of SINDY on increasing numbers of columns.

(b) Behavior of SINDY on increasing numbers of rows.

Figure 5: Behavior of SINDY on increasing numbers of columns and rows.

from there on a large column with JSON data is included that makes up an estimated 50 % of the dataset's overall data volume. For the PDB dataset, the runtime grows seemingly quadratically with the number of columns but settles after around 1,600 columns. The former observation can be explained with the quadratic growth of IND candidates and the latter with the fact that at this point 31 columns per table are used – exactly the number of columns of a large table that contains over 90 % of PDB's data volume.

For the experiments with row scalability, we also read the complete source files and sampled the input rows for SINDY of each file on the fly. Apparently, the algorithm scales linearly with the input data, presumably because using x % of the rows also yields approximately x % of the data volume. In summary, these observations suggest that on typical datasets, the data volume is the main runtime driver for SINDY.

6 Conclusion and Future Work

We presented SINDY, an efficient IND discovery algorithm that can be scaled out on clusters to process large datasets within reasonable time. This scalability is achieved by partitioning the dataset into cells and merging them in a distributed fashion. The result of this merge is then split up into IND candidates that are again merged in a distributed way. We have shown that this algorithm scales well with the volume of a dataset and allows for almost linear scale-out for large datasets. We did not encounter a dataset that could break the algorithm due to its size, even for datasets with almost 2,000 columns. Moreover, SINDY can be quickly adapted to many distributed environments as its logic can be seamlessly implemented on map-reduce-style frameworks like Stratosphere.

Recently, we have extended SINDY to find not only unary but also n-ary INDs using an apriori-based approach like MIND [DLP09]. First experiments have shown that this - even in combination with scale out - is only a limited remedy for the exponential growth of the IND search space. Therefore, our future effort will deal with the question on how to efficiently discover n-ary INDs in scale-out scenarios.

Acknowledgements. This research was funded by the German Research Society (DFG grant no. FOR 1306).

References

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, to appear 2014.

[ASN14] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. DFD: Efficient Discovery of Functional Dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2014.

[BB95] Siegfried Bell and Peter Brockhausen. Discovery of Data Dependencies in Relational Databases. Technical report, Universität Dortmund, 1995.

[BLN06] Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently Computing Inclusion Dependencies for Schema Discovery. In *ICDE Workshops*, 2006.

[BLNT07] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. Efficiently detecting inclusion dependencies. In *ICDE Posters*, 2007.

[CFP82] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion Dependencies and Their Interaction with Functional Dependencies. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1982.

[CTF88] Marco A. Casanova, Luiz Tucherman, and Antonio L. Furtado. Enforcing Inclusion Dependencies and Referencial Integrity. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1988.

[DLP09] Fabien De Marchi, Stéphan Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32:53–73, 2009.

[Gry98] Jarek Gryz. Query folding with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1998.

[HKPT99] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[HQRA+13] Arvid Heise, Jorge-Arnulfo Quiane-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable Discovery of Unique Column Combinations. In *Proceedings of the VLDB Endowment*, 2013.

[LV00] Mark Levene and Millist W. Vincent. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):281–291, 2000.

[RAB+09] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A Machine Learning Approach to Foreign Key Discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, 2009.

[SBHR06] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2006.

[ZHO+10] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On Multi-column Foreign Key Discovery. *Proceedings of the VLDB Endowment*, 3:805–814, 2010.

Image Landmark Recognition with Hierarchical K-Means Tree

Magdalena Rischka, Stefan Conrad

Institute of Computer Science
Heinrich-Heine-University Düsseldorf
D-40225 Düsseldorf, Germany
{rischka, conrad}@cs.uni-duesseldorf.de

Abstract: Today's giant-sized image databases require content-based techniques to handle the exploration of image content on a large scale. A special part of image content retrieval is the domain of landmark recognition in images as it constitutes a basis for a lot of interesting applications on web images, personal image collections and mobile devices. We build an automatic landmark recognition system for images using the Bag-of-Words model in combination with the Hierarchical K-Means index structure. Our experiments on a test set of landmark and non-landmark images with a recognition engine supporting 900 landmarks show that large visual dictionaries of size about 1M achieve the best recognition results.

1 Introduction

Today's giant-sized image databases on the World Wide Web, in personal households and on mobile devices pose great challenges for the user: to manage and use these images in a meaningful way it is necessary to know the images' contents. The contents of web images embedded in web pages can be exploited somehow by analysing the surrounding web text, however images captured by digital cameras or other mobile devices usually include only technical metadata which do not reveal the contents of the image (except GPS data). In these cases the user has to annotate the images with meaningful concepts. Manual content exploration is time-consuming, in such large-scale scenarios even intractable, therefore automatic content-based solutions are required. We focus on the exploration of personal image collections and assume that images have no metadata available. A large amount of the personal collections' images are photos shot in the photographer's vacations and trips showing (prominent) places and landmarks. We address the domain of landmark recognition in images as this topic offers several advantages regarding applications in personal image collections, as well as in the field of the World Wide Web and on mobile devices: the annotation is a basis for a search or can be used as a suggestion for a photo description to the user, the identification of locations by the recognition of landmarks can be used to summarize personal image collections by offering an overview of places the photographer visited. The application of mobile landmark recognition enables tourists to look up sights in real-time to obtain information on them.

Several systems for automatic landmark recognition have been proposed [GBQG09, AKTS10, ZZS+09, PZ08, CBHK09]. Each work addresses different aspects of landmark recognition, the creation of the landmark database, the general problem definition, the initial situation referring metadata (for example if the GPS data of test images is used) and the techniques used, however all of them offer a content-based approach for recognition. We outline these related work with focus on the image representation (features and visual dictionary size in case of Bag-of-Words model or index structures), recognition technique (classifier and/or index structure), dataset (esp. number of landmarks supported) and specifics used. [CBHK09] recognizes images using a Bayesian and a SVM classifier on SIFT-based Bag-of-Words image representation (created with K-Means clustering) with the vocabulary size 1K. The dataset contains 33M self-downloaded Flickr photos and the number of landmarks supported in the evaluation is 10, 25 and 50. [AKTS10] also creates a dataset from Flickr images and then derives scene maps of landmarks which are retrieved with a kd-tree index. The image representation bases on SURF features and a visual dictionary of size 75K. The authors of [GBQG09] create a database from geo-tagged Flickr photos and Wikipedia. The recognition is performed on object-level and with the aid of an approximate K-Means index on SIFT features which constitutes a dictionary of size 500K. [PZ08] uses SIFT features and a Bag-of-Words approach with a dictionary size of 1M. For the dictionary again approximate K-Means index structure is used. The evaluation is performed on *The Oxford Buildings Dataset*[1] (11 landmarks). [ZZS+09] creates the database by crawling travel guide websites (5312 landmarks) and then builds a matching graph out of the feature matches of the images. Images are represented using 118-dim Gabor wavelet texture features with PCA. For retrieval a kd-tree is used, however the size of the underlying dictionary remains unknown. Allmost all presented systems use SIFT features (or the like) with a Bag-of-Words image representation (based on flat K-Means) or a feature index structure like the kd-tree or approximate K-Means. Unfortunately each proposed system is evaluated on different datasets (with different number of landmarks) and different dictionary sizes are used, thus the comparison between methods is difficult. There exists some public databases for landmark recognition, however they are not always suitable for the own problem definition. For example the public datasets *The Paris Dataset*[2] and *The Oxford Buildings Dataset* contain a low number of landmarks (about 12 and 17), which can not be used when developing a landmark engine which has to support thousands of landmarks. Our own experiments [RC14] on different classifiers, dictionary sizes (from 500 to 8K, derived from flat K-Means clustering) and different number of landmarks (45, 300, 600, 900) show that the kNN classifier outperforms the other ones (like SVM) in a large-scale scenario, when the number of landmarks supported increases. Furthermore increasing the dictionary sizes increases the performance. Large (flat) K-Means dictionaries however lead to inefficient recognition times, thus the access to the features/visual words have to be supported by efficient quantizer/index structures. [PJA10] analyses some structured and unstructured quantizer, the families of Locality Sensitive Hashing (LSH) functions as well as the K-Means and the Hierarchical K-Means on SIFT descriptors extracted from a subset of the *INRIA Holidays dataset*[3]. These experiments show that unstructured

[1] http://www.robots.ox.ac.uk/∼vgg/data/oxbuildings/
[2] http://www.robots.ox.ac.uk/∼vgg/data/parisbuildings/
[3] http://lear.inrialpes.fr/people/jegou/data.php

quantizer like the K-Means and the Hierarchical K-Means outperform LSH functions, as they adapt better to the distribution of the features. We develop an automatic content-based image landmark recognition system with a SIFT-based Bag-of-Words image representation in combination with the Hierarchical K-Means (HKM)[NS06] tree index structure and evaluate our system on a large-scale dataset which supports 900 landmarks. Our contribution is to analyse in how far an increasing dictionary up to 3M can improve the recognition performance and whether the HKM tree parameters branching factor and height or only the resulting number of leaf nodes (dictionary size) have an impact on the recognition. Furthermore we analyse the kNN voting distribution of landmark and non-landmark images and based on this we propose a simple and efficient verification filter to decide whether an image contains a landmark or not. In literature this verification is usually done with expensive model fitting algorithms like RANSAC. The remainder of this paper is organized as follows: section 2 introduces into the landmark recognition problem and presents our implemented landmark recognition system with its steps. In section 3 we evaluate our system comparing it to a baseline approach. Finally (section 4) we summarize our results and discuss future work.

2 Automatic Landmark Recognition System

The Landmark Recognition Problem A *landmark* is a physical object, created by man or by nature, with a high recognition value. Usually a landmark is of remarkable size and it is located on a fixed position of the earth. Examples of landmarks are buildings, monuments, statues, parks, mountains and other structures and places. Due to their recognition value, landmarks often serve as geographical points for navigation and localisation. A landmark recognition system has to conduct the following task automatically:

Definition 1 (Landmark Recognition Task)**.** *Given a set of L landmarks $\mathcal{L} = \{l_1, ..., l_L\}$ and an image i which contains the landmarks $\mathcal{G}_i \subseteq \mathcal{L}$. The task for a landmark recognition system is to assign a set of landmarks $\mathcal{P}_i \subseteq \mathcal{L}$ to the image i in such a way that $\mathcal{P}_i = \mathcal{G}_i$.*

$\mathcal{G}_i$ is the *groundtruth set* and $\mathcal{P}_i$ the *prediction set* of the (test) image i. Please note that $\mathcal{G}_i = \emptyset$ and $\mathcal{P}_i = \emptyset$ are possible. The landmark recognition task is a multi-label classification problem including a decision refusal.

The Design of the Landmark Recognition System Our automatic landmark recognition engine bases on a single-label classification approach with the Bag-of-Words image representation and the Hierarchical K-Means tree index structure. This single-label classification approach can be extended to multi-label one (as defined in the Landmark Recognition Task) by applying a postprocessing step which analyses if locally adjacent (ref. city) landmarks (of the classified landmark) are available in the image. This step is not object of this work. Our system consists of two stages: the training stage, in which the recognition engine is learned, and a recognition stage, in which the landmarks in the (test) image are recognized. Figure 1 shows the overall design of our system. The training phase (red path in the figure) begins with the feature extraction step, in which for each training image SIFT features are extracted. Next the HKM tree is created. For this purpose we select a subset

of all SIFT descriptors from the training set and based on this subset we create the HKM tree. The created HKM tree bases only on the SIFT descriptors, still uncoupled from the corresponding training images. For later classification purposes we have to feed the HKM tree with information on the training images. Before the tree feeding step we determine the image representation using the created HKM tree. In the next step the HKM tree is fed with the training images using the information from the training image representation. The resulting HKM tree is ready to use for the recognition phase. In the recognition stage (green path) the test image has to pass the same steps of feature extraction and image representation as the training images. Then the test image is classified to one landmark. In the verification step the classification result is verified, i.e. we check whether the recognized landmark is really available in the image. In this step an assignment of a landmark label to a non-landmark test image should be rejected. The following subsections present each step of the system in detail.

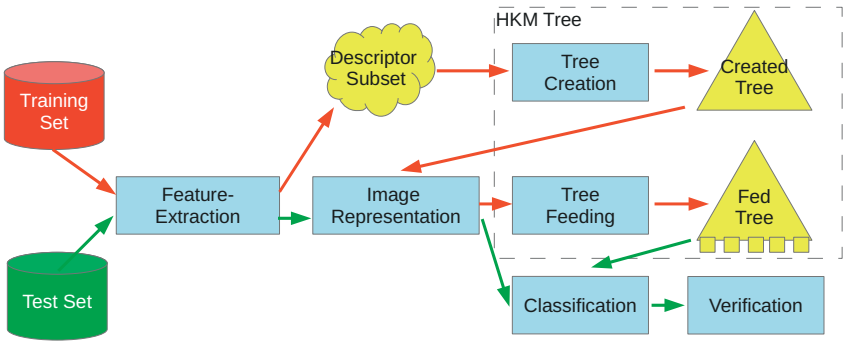


Figure 1: The design of the landmark recognition system

Feature-Extraction We describe images with the local features *Scale Invariant Feature Transform* (SIFT) [Low04]. To extract these features the SIFT algorithm detects stable points in the image and then describes the (small) surrounding area around each point by a histogram of gradients. Finally the SIFT representation of an image i is a set of local SIFT points p: $\text{SIFT}(i) = \{p_1, ..., p_P \mid p = (x, y, s, d)\}$ with x, y are the coordinates of the stable point p in the image, s is the scale influencing the size of the surrounding area and d is the 128-dimensional descriptor which is the histogram of gradients.

Image Representation For the image representation we use the Bag-of-Words (BoW) approach. The idea behind BoW is to aggregate local features to one global descriptor and thus to avoid the expensive comparison of images by matching local descriptors against each other. The BoW descriptor bases on a dictionary of visual words which usually is obtained by partitioning the descriptor-space. Then each partition is represented by an instance of this partition, usually the center of the partition, which is called the *visual word*. A simple and most used method to partition is the use of the K-Means clustering algo-

rithm. However flat K-Means do not scale well with an increasing dictionary size: already dictionary sizes of 10K lead to long recognition times. Applying the HKM tree enables us to use large-sized dictionaries with over 1M words. To create the image representation of a training or test image each SIFT descriptor of the image is looked up in the tree to get the leaf as a visual word. Then the relative frequency of each leaf/visual word in the image is determined. The final image representation is a set of visual words with the corresponding relative frequency.

HKM Tree *General Information* The HKM tree is a tree-based index structure introduced by [NS06]. We use this index structure to maintain the large 128-dimensional SIFT descriptor data of the training images with references to the corresponding training images. The use of an index structure enables fast look-ups of SIFT descriptors and thus an efficient classification of test images at the recognition stage. The HKM tree is created by recursively dividing the data space into disjoint regions using the clustering algorithm K-Means.

Tree Creation For HKM tree creation we take a large-sized and representative subset of all SIFT descriptors of the underlying training set as the root data. For a fixed value K we cluster the root data with K-Means into K clusters. For each resulting cluster consisting of the data part belonging to this cluster we again cluster this data into K clusters. This step is repeated until the data to be clustered reaches a size of $\leq 2K$ data points. This procedure automatically builds a tree of a fixed branching factor K (for each level of the tree) and a tree height h_c (c stands for *created*) determined by the longest clustering path. Each node in the tree corresponds to a data cluster which is representated by a centroid. The tree leaves correspond to the final data partitions.

Tree Feeding In the tree feeding step the relation between the SIFT descriptors and the corresponding training images is established by registering the training images in the leaves of the created HKM tree. To register a training image we take its image representation which is a set of visual words (leaves) with their relative frequencies in this image. The training image is registered in all occurring visual words/leaves of the tree with its relative frequency. Finally each leaf of the HKM tree contains a list of all training images which contain this leaf. To influence the number of leaves of the HKM tree, thus the size of the visual dictionary, we set a height parameter h_f (f stands for *fed*) for the tree which cuts the original created tree and limits its height.

Classification After the test image passed the feature extraction and the image representation step, it is classified to a landmark. For each visited leaf/visual word in the image representation step the training images registered in these leaves are taken for similarity calculation. The similarity between the test image and a training image is determined by the histogram intersection of the relative frequency values of the common leaves. The kNN classifier considers the k nearest training images of the test image. Each of the k nearest training images votes for the landmark it *belongs* to. Thus we can have a maximum of k candidate landmarks for the classification, whereas each candidate landmark can be voted by a maximum of k training images. The number of training images which vote for the finally classified landmark is the *voting score* v. In this work the kNN parameter k is set to 5 as a result of preliminary experiments. We choose a kNN classifier instead of other

well-known classifiers because the kNN has shown classification superiority in large-scale scenarios referring a large number of classes [DBLL10, RC14].

Verification To decide whether a test image contains the classified landmark, we apply the following simple filter based on the kNN classifier:

Definition 2 (Simple kNN Filter)**.** *Given the test image i which has been classified to landmark $\mathcal{P}_i = \{l\}$ with a kNN voting score of v. The classification result of image i is reliable if the voting score v is equal to or exceeds a threshold value $t \in \{1, ..., k\}$, i.e.*

$$i \rightarrow \begin{cases} \emptyset & \text{if } v < t \\ \mathcal{P}_i & \text{if } v \geq t \end{cases} \tag{1}$$

3 Evaluation

Evaluation Dataset For the evaluation we use a self-provided dataset, as there are no public datasets for landmark recognition available which support large number of landmarks. We gathered 900 landmark terms (from 449 cities and 228 countries) from several websites which list landmarks from all over the world, including the website of [ZZS+09][4]. To get images for the training and test sets, we queried the Google image search engine with each landmark term (specified by its region - city or country) and then downloaded the results from the original source. From the downloaded image set we derived three training sets (A, B, C) and a test set. Each training set supports (the same) 900 landmark terms. The test set supports only a subset of the 900 landmark terms: 45 landmark terms (well- and lesser-known landmarks from Europe) with always 20 images per landmark, resulting in 900 *landmark images*. We have choosen the test images manually to ensure their correctness of containing the corresponding landmark and to create a challenging test set: the test images show the landmark in their canonical views, under different perspective changes, distortions and lighting conditions (also at night) as well as indoor shootings and parts of the landmark. To simulate a realistic test set we extended the test set with *non-landmark images*, i.e. images which do not contain any landmark. For this we took a subset of the *Flickr 100k dataset*[5]. All test images have been proofed to be (visually) disjoint from the images of the training sets.

Evaluation Measures To evaluate the performance of the system we use example-based measures which judge the recognition quality for one test image and then we report the average over all test images. In the general problem of multi-label classification we apply example-based Precision and Recall which for an image i with its groundtruth set $\mathcal{G}_i$ and prediction set $\mathcal{P}_i$ are defined as follows:

$$\text{Precision}(i) = \frac{|\mathcal{G}_i \cap \mathcal{P}_i|}{|\mathcal{P}_i|} \qquad \text{Recall}(i) = \frac{|\mathcal{G}_i \cap \mathcal{P}_i|}{|\mathcal{G}_i|} \tag{2}$$

The special cases of Precision and Recall for $\mathcal{G}_i = \emptyset$ and/or $\mathcal{P}_i = \emptyset$ are defined in figure 2. For the restriction to single-label classification, i.e. $|\mathcal{G}_i| = |\mathcal{P}_i| = 1$ Recall and Precision

[4] http://mingzhao.name/landmark/landmark_html/demo_files/1000_landmarks.html
[5] http://www.robots.ox.ac.uk/ vgg/data/oxbuildings/flickr100k.html

have the same value, thus we report only one of them (Recall). Then the average Recall over all test images reports the amount of test images classified correctly, i.e. a landmark image is classified to the correct landmark, a non-landmark image is detected correctly as non-landmark image.

Experiments on the Baseline System In the first experiment we present the results of our baseline approach described in [RC14]. The baseline is a single-label classification approach which uses SIFT features and the Bag-of-Words image representation on a visual dictionary created by (flat) K-Means clustering. For the classifier used, we tested on the 5NN and on an SVM. For the 5NN the baseline system can be seen as a specific case of the HKM tree-based system, in which the HKM tree is obtained by one clustering process resulting in a tree of height 2 (root data level and clustered data level). The Bag-of-Words model has one parameter which is the visual dictionary size. We examine this approach on the following five visual dictionary sizes: 500, 1K, 2K, 4K and 8K. Here larger dictionaries are not considered due to long recognition times using the (flat) K-Means algorithm. Figure 3 shows the recognition results (Recall) of the landmark images (non-landmark images of the test set are not considered) for the classifier 5NN and SVM, before applying the verification step, and depending on the visual dictionary size. Values reported are averages over the three training sets A,B,C. The results reveal that the larger a visual dictionary the better are the classification results. From a dictionary size of 4K on, the 5NN outperforms the SVM and the behaviour of both plots indicate that this can still hold for larger dictionaries. The best recognition result with a Recall value of 0.43 is achieved with a dictionary size of 8K. The trend suggests, that for even larger dictionaries (larger than 8K) the Recall value furthermore grows. This assumption is examined in the next experiment.

K	20	25	15	20	25	15	30	25	30	20	15
h_f	4	4	5	5	5	6	5	6	6	7	8
max	160K	390K	759K	3,2M	9,7M	11M	24M	244M	729M	1,2B	2,5B
real	8T	15T	50T	160T	390T	757T	809T	1M	828T	2,5M	1,4M

Table 1: Maximum ($max = K^{(h_f)}$) and real number of leaves/dictionary sizes of the created HKM trees depending on the branching factor K and the height h_f

Experiments on the HKM Tree-based System This experiment evaluates the HKM tree-based approach on small up to very large dictionaries. To create the HKM tree we take a (representative) subset of 10M SIFT descriptors from the corresponding training set and create trees for the following branching factors K: 15, 20, 25, 30. Then we learn trees (tree feeding step) for different height values, for the height h_c (original created tree) and for smaller heights h_f (tree cuts). Table 1 shows the maximum and the real number of leaves of the trees created depending on the branching factor K and the tree height h_f. Figure 4 shows the recognition results of the test set depending on the parameter visual dictionary size/real number of leaves (x-axis) and the threshold t of the verification step (plots, one color for a fixed threshold). The Recall values reported are averages over the three training sets A,B,C. The diagram on the left of figure 4 shows the 'global' recognition value which is the average over all test images (landmark and non-landmark images). The diagram on

$\mathcal{G}_i$	$\mathcal{P}_i$	Precision	Recall
$\emptyset$	$\emptyset$	1	1
$\emptyset$	$\neq \emptyset$	0	0
$\neq \emptyset$	$\emptyset$	0	0

Figure 2: Definition of Precision and Recall for combinations of $\mathcal{G}_i = \emptyset$ and $\mathcal{P}_i = \emptyset$

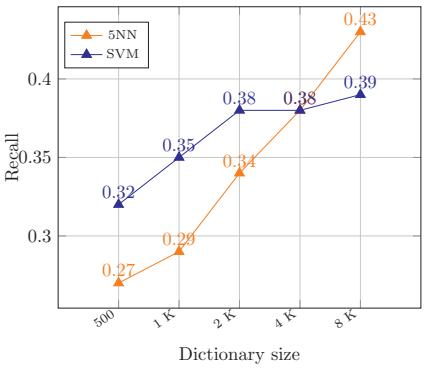


Figure 3: Recognition results of the baseline approach on landmark images

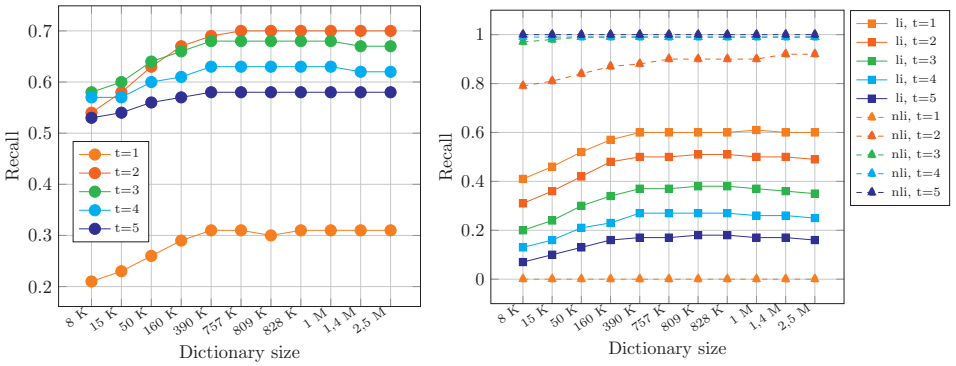


Figure 4: Recognition results of the HKM tree-based approach. Left diagram: on all test images. Right diagram: on landmark (li) and non-landmark images (nli) separately.

the right shows the recognition value for landmark and non-landmark images separately (as averages over all landmark images and all non-landmark images, respectively). The results confirm the assumption derived from the baseline system that larger dictionaries achieve better recognition quality. Up to a dictionary size of 390K the Recall values rise continuously when enlarging the visual dictionary. From a dictionary size of 390K on the recognition value for all test images (left diagram) and for all landmark images (right diagram) levels out and reach the best Recall value of 0.7 and 0.6 on dictionaries of size 800K to 1M for a threshold $t = 2$ and $t = 1$, respectively. Comparing the Recall value of 0.6 ($t = 1$) on landmark images with the best result of the baseline system, we achieve a recognition improvement of 17% by enlarging the dictionary. We can also see that the Recall value for the landmark images, thus for all test images, begins to fall slightly from a dictionary size of 1,4M. However the Recall value of the non-landmark images rises uninterrupted with an increasing dictionary size (for $t > 1$). Please note that for $t = 1$ (equals to no verification step), all test images are assigned to one landmark, thus all non-

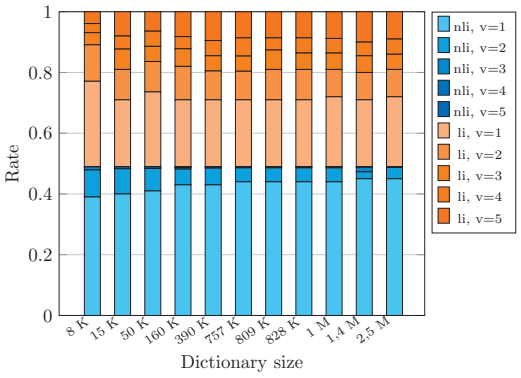


Figure 5: Distribution of the voting score v for $v \in \{1, 2, 3, 4, 5\}$ on landmark (li) and non-landmark images (nli).

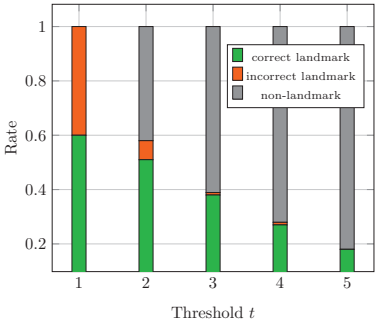


Figure 6: Amount of *correct*, *incorrect* and as *non-landmark* recognized landmark images for a dictionary of size 828K depending on the threshold t.

landmark images are recognized incorrectly and get a Recall value of 0. Due to this fact, the recognition for $t = 1$ on all test images results in a low Recall value. Figure 5 shows the distribution of the voting score v among the whole test set, represented for landmark and non-landmark images separately. The largest amount of test images (concerns both, landmark as non-landmark images) has a voting score $v = 1$. For non-landmark images this amount grows with an increasing dictionary size and is a good indicator for the prediction of non-landmark images. Landmark images are stronger represented with voting scores of $v \geq 2$ than non-landmark images, unfortunately about half of the landmark images still have a voting score of 1. The best trade-off between landmark and non-landmark images according to the global Recall value (left diagram of figure 4) is achieved with $t = 2$. Figure 6 presents the amount of correct, incorrect and as non-landmark recognized landmark images for a dictionary of size 828K depending on the threshold t. The highest rate of correct recognition as well as incorrect recognition is achieved with $t = 1$. Despite the lower amount of correctly recognized landmark images at threshold $t = 2$, this result can be more satisfying for a user than the one at $t = 1$ because the amount of landmark images assigned to the wrong landmark is radically smaller, although simultaneously the amount of landmark images which are predicted as non-landmark is high, too. In a user's perception an incorrect classification (wrong landmark assignment) can be worse than the non-detection (non-landmark prediction) of landmark images. To answer the question on the influence of the branching factor K and the tree height, in figure 4 we can see that the recognition results depend on the real visual dictionary size than on the parameters branching factor and tree height on their own.

4 Summary and Future Work

We developed a CBIR system which bases on the Bag-of-Words approach combined with the HKM tree. Our experiments on different dictionary sizes show that a visual dictionary with a size of about 1M achieves the best recognition results with a Recall value of 0.6 on landmark images and 0.7 on all test images. This improves the baseline approach (dictionary size 8K) by almost 17%. For future work it is interesting to compare other well-known index structures like the kd-tree and the approximate K-Means to see which index structure is most appropriate for the problem of large-scale landmark recognition.

References

[AKTS10] Y. S. Avrithis, Y. Kalantidis, G. Tolias, and E. Spyrou. Retrieving landmark and non-landmark images from community photo collections. In *Proc. of the 18th Int. Conf. on Multimedia 2010, Firenze, Italy*, pages 153–162, 2010.

[CBHK09] D. J. Crandall, L. Backstrom, D. P. Huttenlocher, and J. M. Kleinberg. Mapping the world's photos. In *Proc. of the 18th Int. Conf. on World Wide Web, WWW 2009, Madrid, Spain*, pages 761–770, 2009.

[DBLL10] J. Deng, A. C. Berg, K. Li, and F. F. Li. What Does Classifying More Than 10, 000 Image Categories Tell Us? In *Computer Vision - ECCV 2010 - 11th European Conf. on Computer Vision, Heraklion, Crete, Greece, Proc., Part V*, pages 71–84, 2010.

[GBQG09] S. Gammeter, L. Bossard, T. Quack, and L. J. Van Gool. I know what you did last summer: object-level auto-annotation of holiday snaps. In *IEEE 12th Int. Conf. on Computer Vision, ICCV 2009, Kyoto, Japan*, pages 614–621, 2009.

[Low04] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. Journal of Computer Vision*, 60(2):91–110, 2004.

[NS06] D. Nistér and H. Stewénius. Scalable Recognition with a Vocabulary Tree. In *2006 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR 2006), New York, NY, USA*, pages 2161–2168, 2006.

[PJA10] L. Paulevé, H. Jégou, and L. Amsaleg. Locality Sensitive Hashing: A Comparison of Hash Function Types and Querying Mechanisms. *Pattern Recogn. Lett.*, 31(11):1348–1358, August 2010.

[PZ08] J. Philbin and A. Zisserman. Object Mining Using a Matching Graph on Very Large Image Collections. In *Sixth Indian Conf. on Computer Vision, Graphics & Image Processing, ICVGIP 2008, Bhubaneswar, India*, pages 738–745, 2008.

[RC14] M. Rischka and S. Conrad. Landmark Recognition: State-of-the-Art Methods in a Large-Scale Scenario. In *Proc. of the 16th LWA Workshops: KDML, IR and FGWM, Aachen, Germany.*, pages 10–17, 2014.

[ZZS+09] Y. Zheng, M. Zhao, Y. Song, H. Adam, U. Buddemeier, A. Bissacco, F. Brucher, T. Chua, and H. Neven. Tour the world: Building a web-scale landmark recognition engine. In *2009 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR 2009), Miami, Florida, USA*, pages 1085–1092, 2009.

Erzeugung kalibrierter, metrischer Distanzen mittels multidimensionaler Skalierung

Thomas Böttcher, Ingo Schmitt

Brandenburgische Technische Universität Cottbus – Senftenberg
Walther-Pauer-Str. 2, 03046 Cottbus
tboettcher|schmitt@tu-cottbus.de

Abstract:
Für den Vergleich von Objekten, seien es Texte, Bilder etc, werden in der Regel Ähnlichkeiten bzw. Distanzen bzgl. verschiedener Eigenschaften (z.B. Kanten-, Farb-, Texturfeatures, GPS) genutzt. Werden mehrere Eigenschaften verwendet, führt dies zu einer verbesserten Ausdruckskraft. Problematisch sind hierbei die Eigenschaften der verwendeten Distanzmaße, insbesondere die Dreiecksungleichung. Die Verwendung effizienter Algorithmen, z.B. metrischer Indexsysteme erfordern jedoch diese Eigenschaften. Zusätzlich tritt z.B. bei unterschiedlichen Distanzverteilungen eine Dominanz eines Distanzmaßes auf, die das aggregierte Gesamtergebnis ungewollt verfälscht.
In dieser Arbeit präsentieren wir einen Lösungsansatz, der beide Probleme, mit Hilfe eines Verfahrens der multivariaten Statistik, der multidimensionalen Skalierung (MDS), löst. Wir zeigen wie die Dominanz einer Eigenschaft nachgewiesen und quantifiziert werden kann. Es wird zudem ein erweiterter MDS-Ansatz vorgestellt, der die Vergleichbarkeit verschiedener Distanzmaße gewährleistet. Unser Ansatz erlaubt dabei die Verwendung nicht-metrischer Distanzmaße. Eine Evaluierung auf unterschiedlichen Distanzverteilungen zeigt dabei eine fast vollständige Reduzierung der Dominanz.

1 Einleitung

Im Bereich des Information-Retrievals (IR), Multimedia-Retrievals (MMR), Data-Mining (DM) und vielen anderen Gebieten ist ein Vergleich von Objekten essentiell, z.B. zur Erkennung ähnlicher Objekte oder Duplikate oder zur Klassifizierung der untersuchten Objekte. Der Vergleich von Objekten einer Objektmenge basiert dabei in der Regel auf deren Eigenschaftswerten (Features). Im Bereich des MMR sind Features wie Farben, Kanten oder Texturen häufig genutzte Merkmale. Zur Ausnutzung einer optimalen Ausdruckskraft ist die Verwendung einer geeigneten Kombination verschiedener Features entscheidend.

Der (paarweise) Vergleich von Objekten anhand von Features erfolgt mittels eines Distanz- bzw. Ähnlichkeitsmaßes[1]. Bei mehreren Features lassen sich Distanzen mittels einer Aggregationsfunktion verknüpfen und zu einer Gesamtdistanz zusammenfassen. Problematisch sind hierbei die algebraischen Eigenschaften des Distanzmaßes und die mögliche Dominanz eines Features bzgl. des Aggregationsergebnisses:

1. *Dominanz einer Eigenschaft* - Für einen Ähnlichkeitsvergleich anhand mehrerer Features wird erwartet, dass die Einzelfeature gleichermaßen das Aggregationsergebnis

[1]Beide lassen sich ineinander überführen [Sch06], im Folgenden gehen wir daher von Distanzmaßen aus.

beeinflussen. Häufig gibt es jedoch ein Ungleichgewicht, so dass einzelne Features keinen oder nur geringen Einfluss besitzen.

2. *Nichterfüllung einer Metrikeigenschaft* - Verschiedene Distanzmaße erfüllen unterschiedliche algebraische Eigenschaften und nicht alle Distanzmaße sind für spezielle Probleme gleich geeignet. So erfordern Ansätze zu metrischen Indexverfahren oder Algorithmen im Data-Mining die Erfüllung der Dreiecksungleichung.

Fehlen algebraische Eigenschaften oder gibt es eine zu starke Dominanz, so können die Features und dazugehörigen Distanzmaße nicht mehr sinnvoll innerhalb einer Featurekombination eingesetzt werden.

In bisherigen Arbeiten wurde meist versucht jeweils eines der Probleme zu lösen, eine gemeinsame Betrachtung gibt es bisher nicht [Sko06, WCB06, MA01]. Aus diesem Grund sollen in diesem Paper beide Probleme im Kontext betrachtet werden. Wir werden zunächst ein neues Maß vorstellen, welches die Dominanz eines Features innerhalb der aggregierten Distanz misst. Anschließend soll mit einem angepassten Verfahren der multivariaten Statistik, der multidimensionale Skalierung (MDS), gezeigt werden, wie die genannten Probleme gelöst werden können. Mittels einer Erweiterung des rang-erhaltenden MDS-Verfahrens kann gezeigt werden, dass fehlende Metrikeigenschaften, insbesondere die Dreiecksungleichung, kompensiert werden können. Eine Evaluierung auf unterschiedlichen Distanzverteilungen zeigt dabei eine starke Reduktion der Dominanz.

Die Arbeit ist dabei wie folgt aufgebaut. In Kapitel 2 werden grundlegende Definitionen von Distanzmaßen, Aggregation und Dominanz eines Features erläutert. Kapitel 3 liefert einen Überblick über den Stand der Technik. Kapitel 4 erläutert die Grundlagen der MDS und ihre Grenzen. Kapitel 5 beschreibt die notwendigen Erweiterungen der MDS zur Lösung genannter Probleme. Eine Evaluierung der Ergebnisse erfolgt in Kapitel 6. Kapitel 7 gibt eine Zusammenfassung sowie einen Ausblick über zukünftige Arbeiten.

2 Grundlagen

Das folgende Kapitel definiert die grundlegenden Begriffe und die Notationen, die in dieser Arbeit verwendet werden.

2.1 Grundlegende Definitionen

Eine Ähnlichkeitsberechnung erfolgt in der Regel auf der Grundlage einer Objektmenge $O \subseteq \mathbb{U}$ im Universum $\mathbb{U}$. Ein Distanzmaß zwischen zwei Objekten basierend auf einem Feature p sei als eine Funktion $d : \mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}_{\geq 0}$ definiert. Zur Klassifikation der unterschiedlichen Distanzmaße werden folgende vier Eigenschaften genutzt: Selbstidentität: $\forall o \in O : d(o,o) = 0$, Positivität: $\forall o_r \neq o_s \in O : d(o_r,o_s) > 0$, Symmetrie: $\forall o_r,o_s \in O : d(o_r,o_s) = d(o_s,o_r)$ und Dreiecksungleichung: $\forall o_r,o_s,o_t \in O : d(o_r,o_t) \leq d(o_r,o_s) + d(o_s,o_t)$. Erfüllt eine Distanzfunktion alle vier Eigenschaften, so wird sie als Metrik bezeichnet [Sam06], ohne die Eigenschaft der Dreiecksungleichung wird sie als Semidistanzfunktion bezeichnet.

Für eine Distanzberechnung mit m Features $p = (p_1,\ldots,p_m)$ werden zunächst die partiellen Distanzen $\delta^j_{rs} = d^j(o_r,o_s)$ bestimmt. Anschließend werden die partiellen Distanzwerte δ^j_{rs} mittels einer Aggregationsfunktion $agg : \mathbb{R}^m_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$ zu einer Gesamtdistanz aggregiert.

Die Menge aller aggregierten Distanzen für Objektpaare aus O mit $|O| = n$ sei als $\delta^j = \{\delta^j_1, \delta^j_2 \ldots, \delta^j_l\}$ mit $l = \frac{n^2-n}{2}$ bestimmt.

2.2 Dominanz eines Features

Für einen Ähnlichkeitsvergleich von Objekten anhand mehrerer Merkmale sollen die Einzelmerkmale gleichermaßen das Aggregationsergebnis beeinflussen. Offen ist dabei, wann eine Dominanz eines Features auftritt und wie der Grad der Dominanz gemessen werden kann. Betrachtet man in ein festes Intervall (z.B. $[0,1]$) normierte Distanzen, so kann eine unterschiedliche Distanzverteilung einer Stichprobe, insbesondere unterschiedliche Intervallgrößen, zu einer Dominanz eines Features führen [BS14]. Wir definieren den Grad der Dominanz mittels eines Korrelationsmaßes, wobei dieser als Abweichung zum gewünschten Zustand (kalibrierte Distanzverteilung) gemessen wird [BS14, Kub12]:

$$Cal_{err}(\delta^i, \delta^j, \delta^{agg}) = 1 - \frac{4}{\pi}\arctan\left(\frac{Corr(\delta^j, \delta^{agg})}{Corr(\delta^i, \delta^{agg})}\right). \quad (1)$$

Hierbei definiert $Corr(X,Y)$ ein geeignetes Korrelationsmaß, in unserem Fall genügt der Rangkorrelationskoeffizient von Spearman [Spe04]. Es wird immer die Kalibrierung von zwei Verteilungen bezüglich des Aggregationsergebnisses betrachtet. Die Korrelationswerte lassen sich als Punkt in $[0,1]^n$ auffassen. Die Verteilungen sind kalibriert ($Cal_{err} = 0$), wenn sie gleich stark mit dem aggregierten Wert korrelieren, d.h. auf der Winkelhalbierenden liegen. Bei allen Punkten unterhalb dieser Linie dominiert die erste Verteilung das Aggregationsergebnis, bei allen oberhalb die zweite Verteilung.

3 Stand der Technik

Das Problem fehlender Metrikeigenschaften, insbesondere der Dreiecksungleichung, ist bereits aus verschiedenen Gebieten bekannt. Skopal [Sko06] stellt mit seinem Tri-Gen-Algorithmus einen Ansatz bereit, um eine Semi-Metrik in eine Metrik zu transformieren. Hierzu wird eine Teilmenge der Daten und deren Distanzverteilung genutzt. Ist die Teilmenge jedoch zu klein, kann nicht sichergestellt werden, dass der Tri-Gen-Ansatz eine vollständige Metrik liefert. Einen zur MDS sehr ähnlichen Ansatz stellt FastMap [FL95] dar. Dieser Ansatz bildet Daten im k-dimensionalen Raum ab. Durch die Verwendung einer L_p-Norm können auch die Eigenschaften einer Metrik sichergestellt werden. FastMap erreicht dabei eine verbesserte Laufzeit $O(n \log n)$ gegenüber $O(n^2)$ bei der MDS. Die Reihenfolge der Distanzen wird bestmöglich erhalten - eine Garantie gibt es jedoch nicht.

Score-Normalization stellt einen zweiten wichtigen Themenbereich dieser Arbeit dar. Die Evaluierung solcher Ansätze erfolgt in vielen Fällen, vor allem im Bereich des IR, direkt über die Auswertung der Qualität der Suchergebnisse. Dieses Vorgehen liefert aber kaum Anhaltspunkte, warum sich einige Normalisierungsansätze besser für bestimmte Anwendungen eignen als andere [Kub12]. Eine Übersicht über die zahlreichen Ansätze (nicht-)linearer Normalisierungen findet sich in [WCB06, Kub12, BS14]. Für das Problem der Dominanz lässt sich einfach zeigen, dass diese Ansätze keinen direkten Einfluss auf die Distanzverteilung haben. Es werden maximal zwei statistische Merkmale (Minimum, Maximum, Median etc.) genutzt, um eine Normalisierung durchzuführen [MA01]. Besonders

problematisch ist die mangelnde Robustheit gegenüber Ausreißern.

Ansätze, die Distanzverteilung als Grundlage zur Normalisierung heranziehen, versuchen Distanzen aus unterschiedlichen Quellen so abzubilden, dass sie möglichst exakt gleiche Verteilungen besitzen. Die Ansätze von Manmatha [MRF01] und Fernandez [FVC06] analysieren dabei das probabilistische Verhalten von Suchmaschinen unter der Annahme, dass relevante Dokumente eine Normalverteilung und irrelevante eine exponentielle Verteilung besitzen. Diese Ansätze bieten zwar eine optimierte Normierung, erfordern aber gleichzeitig Informationen über die Relevanz von Dokumenten, die häufig nicht vorhanden sind.

4 Multidimensionale Skalierung

Die Multidimensionale Skalierung (MDS) ist ein Verfahren der multivariaten Statistik und dient ursprünglich dem Vergleich einer Vielzahl von Objekten oder Datenpunkten anhand eines Nachbarschaftsmaßes [GR72]. Mittels MDS wird versucht, eine Menge von Objekten O mit Hilfe der paarweisen Distanzen in einem t-dimensionalen Raum anzuordnen [BG05]. Eine solche Anordnung bezeichnen wir als *Konfiguration*. In einigen Fällen wird die MDS auch zur Visualisierung der Daten ($t = 2$) eingesetzt.

In diesem Kapitel soll erläutert werden, welche Anforderungen die MDS erfüllen muss, um die Probleme der Dreiecksungleichung und der Kalibrierung lösen zu können. Anschließend werden das Prinzip und die Grenzen der nicht-metrischen MDS beschrieben.

4.1 Anforderungen

Zur Lösung des Problems fehlender Dreiecksungleichung muss aus einer nicht-metrischen Ausgangsdistanzmatrix D eine metrische Distanzmatrix D' erzeugt werden. Das Verfahren der Multidimensionalen Skalierung arbeitet mit Konfigurations- und Distanzmatrizen. Die durch die MDS erzeugte Konfiguration besteht aus Vektoren im mehrdimensionalen euklidischen Raum. Wir müssen zunächst zeigen, dass die aus der Konfiguration abgeleitete Distanzmatrix D' die Eigenschaften einer Metrik, insbesondere die der Dreiecksungleichung erfüllt. Zusätzlich muss nachgewiesen werden, dass die Rangfolge der Ursprungsdistanzen erhalten bleibt, d.h. $\text{Rang}(D(o_r, o_s)) = \text{Rang}(D'(o_r, o_s))$.

Für die Lösung des Dominanzproblems müssen die erzeugten Distanzmatrizen gleichermaßen das Aggregationsergebnis beeinflussen. Erzeugte Distanzverteilungen von D'_j sollten daher unabhängig von der Eingangsdistanzmatrix D_j möglichst identisch[2] sein.

4.2 Nicht-Metrische Multidimensionale Skalierung

Der nicht-metrische Ansatz der MDS nach Kruskal [Kru64a] arbeitet ausschließlich auf den Rängen der Distanzmatrix. Ausgangspunkt der MDS ist eine symmetrische[3] Distanzmatrix D bestehend aus jeweils paarweisen Distanzen δ_{ij}. Sei $\hat{D}$ die obere Dreiecksmatrix von D. Wir definieren nun ausgehend von $\hat{D}$ die zu erhaltende Rangfolge:

[2] Identisch bezieht sich hierbei auf die Größe des Intervalls in dem die Distanzen verteilt sind, so dass die Korrelation zu den Aggregationsdistanzen gleich ist.

[3] Eine Transformation einer nicht-symmetrischen in eine symmetrische Distanzmatrix ist nicht Bestandteil dieser Arbeit. Ein möglicher Ansatz findet sich z.B. in [Kru64a].

$$\hat{\delta}_1 \leq \hat{\delta}_2 \leq \cdots \leq \hat{\delta}_K, \tag{2}$$

wobei $\hat{\delta}_1$ hierbei der kleinsten Distanz $\delta_{ij} \in \hat{D}$ entspricht.

Die Repräsentation der Objekte im t-dimensionalen Raum erfolgt mittels Konfigurationen. Eine Konfiguration für n Objekte ist nach Kruskal durch $C = (x_1, \ldots, x_n), x_i \in \mathbb{R}^t$ definiert. Für die Erzeugung einer solchen Konfiguration genügen die Ränge der Distanzwerte. Eine initiale Konfiguration C wird typischerweise zufällig erzeugt. Zur Erhaltung der ursprünglichen Rangfolge müssen die aus der Konfiguration abgeleiteten Distanzen d_{ij} die gleiche Rangfolge ergeben. Um dies zu gewährleisten, definieren wir folgende Monotoniebedingung für alle Objektpaare:

$$\delta_{ij} > \delta_{kl} \Rightarrow d_{ij} > d_{kl}. \tag{3}$$

Die Distanzen d_{ij} werden hierbei durch die Verwendung der L_2-Norm bestimmt[4].

Um eine Konfiguration bzgl. (3) bewerten zu können, nutzen wir den von Kruskal eingeführten Stresswert $S = (\sum_{i<j}(d_{ij} - \hat{d}_{ij})^2 / \sum_{i<j} d_{ij}^2)^{\frac{1}{2}}$, wobei d_{ij} die Konfigurationsdistanzen und $\hat{d}_{ij}$ die Disparitäten zwischen den Objekten o_i und o_j [Kru64a] sind. Eine optimale Konfiguration (Erfüllung von (3)) ist bei einem Stresswert $S = 0$ erreicht. Ist die Monotoniebedingung zwischen zwei benachbarten Distanzen gemäß (3) verletzt ($S > 0$), so erfolgt eine Anpassung der Distanzen als Disparitäten mittels Pool-Adjacent Violators (PAV) [RDJH73, ABE+55]. Dabei wird zunächst für die benachbarten Distanzen ein Pool gebildet und beide Distanzen durch den Mittelwert ersetzt. Wurde dadurch eine weitere Verletzung zu der vorherigen Distanz erzeugt, so wird diese ebenfalls in den Pool aufgenommen und der Mittelwert bestimmt. Dies wird solange fortgesetzt, bis keine Monotonieverletzung mehr vorliegt. Allerdings gibt es bei diesem Verfahren häufig Distanzen mit gleichem Wert.

Da in der Regel die initiale Konfiguration nicht genügt, muss diese modifiziert werden. Das Finden einer optimalen Konfiguration ist nicht trivial, da es möglich ist, dass mehrere oder (im Fall $t \ll n$) keine Lösungen existieren können, die (3) erfüllen. Um eine näherungsweise Lösung zu finden, schlägt Kruskal einen iterativen Prozess vor. Erhalten wir nach der Stressberechnung $S < \epsilon$, wobei ϵ ein vordefinierter maximaler Stresswert ist, so ist eine Zielkonfiguration gefunden. Andernfalls erfolgt die Berechnung einer modifizierten Konfiguration $\hat{C}$ aus den Koordinaten x_{it} (Wert von Objekt o_i in Dimension t). Hierfür werden zwei Objekte o_i und o_j genutzt, um die Position von o_i in Bezug auf o_j in der Dimension r durch $x'_{ir} = x_{ir} + \alpha(1 - \frac{\hat{d}_{ij}}{d_{ij}}) \cdot (x_{jr} - x_{ir})$, für $i \neq j, r = (1, \ldots, t)$ neu zu berechnen [Kru64b]. Eine Neupositionierung des Objektes o_i bezüglich aller anderen Objekte innerhalb einer Dimension r kann wie folgt abgeleitet werden:

$$x'_{ir} = x_{ir} + \frac{\alpha}{n-1} \sum_{j=1}^{n} (1 - \frac{\hat{d}_{ij}}{d_{ij}}) \cdot (x_{jr} - x_{ir}),\ r = (1, \ldots, t). \tag{4}$$

Der Parameter α definiert eine Schrittweite, zur Anpassung der Koordinaten von x_{ir}. Ein kleinerer Wert erlaubt eine höhere Genauigkeit der Lösung, jedoch auf Kosten von mehr Iterationen [Kru64a].

[4]Die Verwendung nicht-euklidischer Distanzmaße wird in [Kru64a] diskutiert.

4.3 Grenzen der nicht-metrischen MDS

Für die Erzeugung einer metrischen Distanzmatrix wird die aus der MDS erzeugte Konfiguration C genutzt. Es gibt hierbei zwei Lösungsansätze. Es können die aus den Konfigurationen bestimmten Distanzen d_{ij} direkt genutzt werden. Alternativ kann aus der Konfiguration eine positiv semidefinite Ähnlichkeitsmatrix $A = CC^T$ erzeugt werden. Die mittels des Skalarprodukts erzeugte Ähnlichkeitsmatrix kann anschließend rangerhaltend in eine metrische Distanzmatrix transformiert werden. Problematisch ist hierbei, dass unter Verwendung von Gleichung (4) die Objekte einer Konfiguration beliebig im t-dimensionalen Raum verteilt sein können und somit die maximal mögliche Distanz zwischen zwei Objekten nicht bestimmt werden kann. Hinzu kommt, dass die Verwendung des Pool-Adjacent Violator-Verfahrens nur eine schwache Monotonie zwischen Distanzen und Disparitäten beinhaltet, d.h. $d_{ij} > d_{kl} \Rightarrow \hat{d}_{ij} \geq \hat{d}_{kl}$ sowie $\delta_{ij} > \delta_{kl} \Rightarrow \hat{d}_{ij} \geq \hat{d}_{kl}$. Zur Erzeugung einer linearen Abbildung der Distanzen unabhängig von den Ursprungsdaten benötigen wir jedoch eine strikte Ordnung von Distanzen und Disparitäten: $\delta_{ij} > \delta_{kl} \Rightarrow \hat{d}_{ij} > \hat{d}_{kl}$. Zusätzlich müssen die Distanzen auf ein festes Intervall beschränkt werden. Der bisherige PAV-Algorithmus genügt somit nicht für die Lösung des Dominanzproblems und muss daher erweitert werden.

5 Erweiterung der Multidimensionalen Skalierung

Im vorherigen Kapitel haben wir gezeigt, dass die nicht-metrische MDS in der ursprünglichen Form nicht ausreicht, um unsere definierten Probleme lösen zu können. Im folgenden Kapitel werden mehrere Anpassungen an das ursprüngliche Verfahren erläutert. Anschließend zeigen wir, dass diese Modifikationen genügen, um sowohl das Problem der Dreiecksungleichung als auch das Problem der Dominanz zu lösen.

5.1 Konfigurationsnormalisierung

Da die ursprüngliche Konfigurationsberechnung für das Problem der Dreiecksungleichung nicht geeignet ist, wird das MDS-Verfahren durch eine Konfigurationsnormalisierung erweitert. Wir werden hierzu jeden Punkt x_i der Konfiguration C auf die Oberfläche der Hyperkugel mit einem Radius $r = 1$ und t Dimensionen projizieren. Als initiale Konfiguration wurde bisher eine zufällige Verteilung der Objekte in $\mathbb{R}^t$ angenommen. O.B.d.A. können wir die initiale Konfiguration auch auf den ersten Orthanten beschränken, d.h. alle Koordinaten sind stets positiv. Für eine gegebene Konfiguration C definieren wir die normalisierte Konfiguration als $C' = (x_1', \ldots, x_n'), x_i' \in \mathbb{R}^t$, wobei $x_i' = \frac{x_{it}}{l_i}$ und $l_i = \sqrt{x_{i1}^2 + \cdots + x_{it}^2}$. Betrachten wir die Koordinaten eines Objektes o_i als Vektor v_i, so erhalten wir stets einen Vektor der Länge 1 ($|v_i| = 1$). Zusätzlich wird die maximale Distanz zwischen zwei Objekten beschränkt, d.h. $\forall o_i, o_j : d(o_i, o_j) \leq \sqrt{2}$.

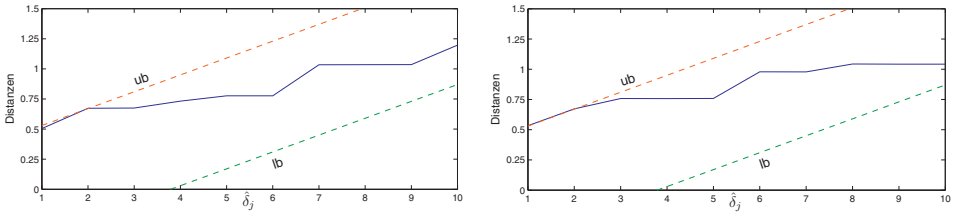
5.2 Erweiterung des PAV-Algorithmus

Um die Vergleichbarkeit erzeugter Distanzmatrizen zu gewährleisten, genügt die Normalisierung der Konfiguration nicht. Hierzu erweitern wir das PAV-Verfahren [RDJH73], um einen linearen Anstieg der Distanzwerte (vgl. Abbildung 1) zu erzeugen. Wir definieren

zunächst einen festen Korridor, in denen die Konfigurationsdistanzen und Disparitäten liegen dürfen. Anschließend erzeugen wir eine totale Ordnung ($\delta_{ij} > \delta_{kl} \Rightarrow \hat{d}_{ij} > \hat{d}_{kl}$) auf den Disparitäten, so dass keine Disparitäten mehrfach auftreten. Um die Konfigurationsdistanzen innerhalb eines festen Korridors zu beschränken, wird ein Bereich (*range*) definiert, der sicherstellt, dass unabhängig von den Eingangsdaten alle Konfigurationsdistanzen auf das gleiche Intervall abgebildet werden. Hierzu wird eine obere Grenze $ub = min(\sqrt{2}, middle + \frac{range}{2})$ sowie eine untere $lb = max(0, middle - \frac{range}{2})$ definiert, wobei $middle = (ic - 0.5) \cdot \frac{\sqrt{2}}{K}$ ist. K definiert die Anzahl der Distanzen und ic ist ein Iterationszähler beim Erzeugen der Pools, um Distanzen gleichmäßig über das Intervall zu verteilen. Sollten Distanzen außerhalb der Grenzen (*ub* oder *lb*) liegen, wird ihnen der Grenzwert zugeordnet. Die Größe des Intervalls (*range*) liegt im Bereich $[0, \sqrt{2}]$ und variiert mit der Anzahl der Objekte.

Zur Erzeugung einer totalen Ordnung werden alle Distanzen eines Pools mit einem festen Wert inkrementiert. Sei $p(m)$ der m-te PAV-Pool mit $size(p(m))$ Elementen. Weiterhin sei $dist(p(m))$ der Distanzwert eines Pools m. Der Basiswert für die Inkrementierung für einen Pool sei durch $inc(m) = \frac{dist(p(m+1)) - dist(p(m))}{size(p(m))}$ bestimmt. Mit Hilfe des Basiswertes (*inc*) wird nun ein linearer Verlauf der Distanzen von einem Pool zum Nächsten ermöglicht. Sei $p(m)(i)$ die i-te Position im Pool, dann berechnet sich die i-te Disparität von Pool m durch $disparity(m,i) = dist(p(m)) + inc(m) \cdot (i-1), \forall i = 1, \ldots, size(p(m))$.

Abbildung 1 zeigt den erzeugten linearen Verlauf der Konfigurationsdistanzen gegenüber den Ausgangsdistanzen. Die Ursprungsdaten entsprechen dem Beispiel aus [BS14] und weisen initial durch zwei unterschiedliche Distanzverteilungen einen Kalibrierungsfehler von $Cal_{err} = 0.43$ auf. Unter Verwendung des erweiterten MDS-Algorithmus reduziert sich bei korrekter Erhaltung der Ursprungsrangfolge der Kalibrierungsfehler auf $Cal'_{err} = 0,01$.



(a) ν_1 (b) ν_2

Abbildung 1: Verteilung der Konfigurationsdistanzen unter Verwendung des erweiterten MDS-Algorithmus ($\alpha = 0.2, range = 0.7, t = 4$)

5.3 Zusammenfassung

Mit der Erweiterung des iterativen MDS-Verfahrens können wir nun eine Konfiguration erzeugen, die einerseits das Problem der Dreiecksungleichung und anderseits das Dominanzproblem löst.

Ausgangspunkt für die Erzeugung einer metrischen Distanzmatrix ist eine normalisierte Konfiguration C. Betrachtet man jedes Objekt als einen normierten Zeilenvektor $v_i =$

$(x_1, \ldots, x_t)$, kann eine normierte, positiv semidefinite Ähnlichkeitsmatrix $A = CC^T$ erzeugt werden, wobei die Ähnlichkeit zwischen zwei Objekten dem Skalarprodukt, also dem Kosinus des eingeschlossenen Winkels entspricht. Das Skalarprodukt von v_i zu sich selbst ist stets 1. Alle anderen Werte liegen im Intervall $[0, 1]$. Mit dem Kosinussatz kann aus dem Ähnlichkeitswert a_{ij} unter automatischer Erfüllung der Dreiecksungleichung eine Distanz $d'(i,j) = \sqrt{2 - 2\,a_{ij}}$ abgeleitet werden.

Erhalten wir durch die MDS eine Konfiguration mit einem Stresswert von 0 bzw. bei einem hinreichend kleinen ϵ mit $S < \epsilon$, so gilt auch (3). Damit wird eine exakte Abbildung der Ursprungsrangfolge durch die Konfiguration C gewährleistet.

Durch Anpassung des PAV-Verfahrens kann mittels MDS eine Konfiguration erzeugt werden, deren abgeleitete Distanzen gleichmäßig über ein definiertes Intervall verteilt sind. Dies ermöglicht es, Distanzmatrizen zu erzeugen, die unabhängig von ihren Ursprungsdaten das Aggregationsergebnis gleichmäßig beeinflussen. Das Beispiel aus Abbildung 1 zeigt exemplarisch, dass die Dominanz nahezu vollständig beseitigt wurde. Eine umfangreichere Evaluierung erfolgt im nächsten Kapitel. Offen für zukünftige Arbeiten bleiben zunächst Lösungsansätze zur automatisierten Findung der Parameter α, $range$ und t.

6 Evaluierung

In diesem Kapitel soll der Grad der Dominanz mit Hilfe des Kalibrierungsfehlers evaluiert werden. Wir werden zunächst die Auswirkungen des erweiterten MDS-Verfahrens zeigen. Anschließend erfolgt ein Vergleich mit zwei häufig eingesetzten Normalisierungsverfahren. Zur Durchführung der Experimente wurden Matlab-Skripte implementiert.

Durch die Erzeugung einer Konfiguration mittels des erweiterten MDS-Algorithmus wird eine nahezu lineare Abbildung von Distanzen ermöglicht (vgl. Abbildung 1). Wir zeigen nun, dass die erzeugten Distanzverteilungen unabhängig von den Ausgangsdaten ähnlich in Bezug auf die Korrelation zum Aggregationsergebnis sind. Zur Evaluierung wurden in 1000 Durchläufen verschiedene Objektmengen ($n = 5, 10, 30, 50, 100$) aus Normal- und Gleichverteilungen normalisiert. Es zeigte sich, dass der Kalibrierungsfehler fast vollständig entfernt wurde und somit das Problem der Dominanz nach der Kalibrierung durch die MDS nicht mehr vorhanden ist. Tabelle 1 zeigt einige Auszüge der Ergebnisse, wobei neben den MDS-Parametern t, α und range auch die Stresswerte S_1 und S_2 nach der MDS sowie die Korrelationswerte ρ_1 und ρ_2 und die Kalibrierungsfehler vor und nach der Normierung dargestellt sind. Durchschnittlich konnte der Kalibrierungsfehler um 94% reduziert werden. Dies wird zudem anhand der Korrelationswerte ρ_i vor und nach der MDS deutlich.

Tabelle 1: Veränderung des Kalibrierungsfehlers nach Anwendung der MDS

Obj.	t	α	**range**	S_1	S_2	ρ_1	ρ_2	Cal_{err}	ρ_1'	ρ_2'	Cal_{err}'
5	4	0,22	0,95	0,0032	0,0072	0,939	0,454	**0,426**	0,757	0,696	**-0,053**
5	4	0,22	0,95	0,0004	0,0046	0,963	0,575	**0,314**	0,878	0,866	**0,008**
10	8	0,22	0,95	0,0112	0,0272	0,958	0,169	**0,777**	0,701	0,638	**0,063**
30	15	0,15	1,15	0,076	0,0833	0,928	0,294	**0,61**	0,659	0,667	**-0,008**
100	35	0,025	1,25	0,0746	0,0787	0,933	0,344	**0,55**	0,66	0,684	**0,022**

In einer zweiten Evaluierung sollen die Auswirkungen auf das Dominanzproblem genauer

untersucht werden. Hierzu vergleichen wir unseren Ansatz mit der Min/Max-[WCB06] und der Z-Score-Normalisierung [MA01]. Grundlage der Evaluierung bilden jeweils drei zufällig erzeugte Distanzverteilungen (Normalverteilung, Gleichverteilung und ein Mischverteilung aus Normal- und Gleichverteilung). Die hierfür genutzten Distanzen basierten auf zwei unterschiedlich großen Intervallen ($[0.2, 0.3]$ vs. $[0.2, 0.9]$). Darüber hinaus soll die Anfälligkeit gegenüber Ausreißern geprüft werden. Hierzu wurden der Verteilung mit dem kleinen Intervall (die dominierte Verteilung) künstlich Distanzen außerhalb des eigentlichen Intervalls hinzugefügt, um die Dominanz künstlich zu reduzieren. Die Verteilung enthielt dabei einen verschmutzten Anteil von 0.5% aller Distanzen. Abbildung 2 zeigt die Ergebnisse der Evaluierung. Zur Messung der Streuung wurden die Experimente 1000 mal wiederholt und der durchschnittliche Kalibrierungsfehler nach der Normalisierung gemessen. Besonders bei Verteilungen mit Ausreißern und bei unterschiedlichen Verteilungen zeigt der erweiterte MDS-Ansatz seine Vorteile. Leichte Abweichungen in den Ergebnissen des MDS-Verfahrens lassen sind durch die zufällig erzeugten Distanzen und die möglicherweise nicht optimal gewählten MDS-Parameter erklären (vgl. Kapitel 4.2). Dies und die Tatsache, dass Z-Score-Normalisierung beste Ergebnisse bei einer Normalverteilung erzielt, erklären auch den minimal höheren Kalibrierungsfehler bei Experimenten mit Normalverteilung ohne Outlier. Insgesamt kann das Verfahren dennoch als robust gegenüber Ausreißern oder verschiedenen Verteilungen bezeichnet werden.

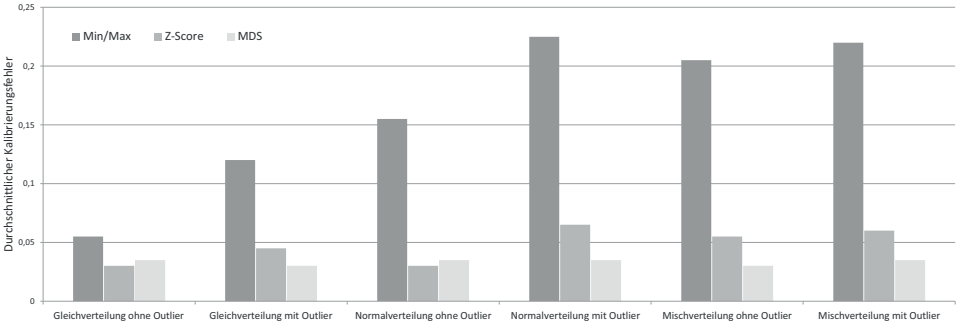


Abbildung 2: Durchschnittlicher Kalibrierungsfehler nach der Normalisierung

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Ansatz vorgestellt, der das Problem fehlender Metrikeigenschaften sowie der Dominanz eines Features bzgl. einer Aggregation mit Hilfe einer Erweiterung der multidimensionalen Skalierung löst. Es konnte gezeigt werden, dass unter Erhaltung der Reihenfolge eine Distanzmatrix erzeugt werden kann, die alle Eigenschaften einer Metrik erfüllt, insbesondere der Dreiecksungleichung. Weiterhin konnte gezeigt werden, dass die MDS auch das Dominanzproblem reduzieren und im optimalen Fall beseitigen kann.

In zukünftigen Arbeiten soll das Verfahren vor allem im Hinblick auf die Laufzeit weiter verbessert werden. Eine Kombination mit dem MDS-nahen Ansatz *FastMap* könnte eine effizientere Berechnung auch auf größeren Datenmengen ($n > 10000$) ermöglichen.

Literatur

[ABE+55] Miriam Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid und Edward Silverman. An Empirical Distribution Function for Sampling with Incomplete Information. *Ann. Math. Statist.*, 26(4):641–647, 1955.

[BG05] I. Borg und P.J.F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 2005.

[BS14] Thomas Böttcher und Ingo Schmitt. Dominanzproblem bei der Nutzung von Multi-Feature-Ansätzen. In *Grundlagen von Datenbanken*, 2014.

[FL95] C. Faloutsos und K. Lin. FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. In *Proceedings of 1995 ACM SIGMOD*, Seiten 163–174, 1995.

[FVC06] Miriam Fernández, David Vallet und Pablo Castells. Probabilistic Score Normalization for Rank Aggregation. In Mounia Lalmas, Andy MacFarlane, Stefan M. Rüger, Anastasios Tombros, Theodora Tsikrika und Alexei Yavlinsky, Hrsg., *Advances in Information Retrieval, 28th European Conference on IR Research, ECIR 2006, London, UK, April 10-12, 2006, Proceedings*, Jgg. 3936 of *Lecture Notes in Computer Science*, Seiten 553–556. Springer, 2006.

[GR72] Paul E. Green und Vithala R. Rao. *Applied multidimensional scaling; a comparison of approaches and algorithms*. Holt, Rinehart and Winston New York, 1972.

[Kru64a] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, Marz 1964.

[Kru64b] J.B. Kruskal. Nonmetric multidimensional scaling: A numerical method. *Psychometrika*, 29(2):115–129, 1964.

[Kub12] Robert Kuban. Analyse von Kalibrierungsansätzen für die CQQL-Auswertung. Bachelor's thesis, University of Cottbus-Senftenberg, Germany, 10 2012.

[MA01] Mark Montague und Javed A. Aslam. Relevance Score Normalization for Metasearch. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, Seiten 427–433, New York, NY, USA, 2001. ACM.

[MRF01] R. Manmatha, T. Rath und F. Feng. Modeling score distributions for combining the outputs of search engines. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, Seiten 267–275, New York, NY, USA, 2001. ACM Press.

[RDJH73] Barlow R.E., Bartholomew D.J., Bremner J.M. und Brunk H.D. Statistical inference under order restrictions. *Statistica Neerlandica*, 27(4):189–189, 1973.

[Sam06] Hanan Samet. *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann, 2006/08/08/ 2006.

[Sko06] Tomás Skopal. On Fast Non-metric Similarity Search by Metric Access Methods. In *EDBT*, Seiten 718–736, 2006.

[Spe04] C. Spearman. The Proof and Measurement of Association Between Two Things. *American Journal of Psychology*, 15:88–103, 1904.

[WCB06] Shengli Wu, Fabio Crestani und Yaxin Bi. Evaluating Score Normalization Methods in Data Fusion. In *Proceedings of the Third Asia Conference on Information Retrieval Technology*, AIRS'06, Seiten 642–648, Berlin, Heidelberg, 2006. Springer-Verlag.

Big Data-Zentren – Vorstellung und Panel

Big Data-Zentren – Vorstellung und Panel

Volker Markl (BBDC), Erhard Rahm & Wolfgang Lehner (ScaDS), Michael Beigl (SDIL)

Moderation: Thomas Seidl

1 Abstract

Zur Erforschung der verschiedenen Facetten von „Big Data“ wurden jüngst drei Zentren gegründet. Hierbei handelt es sich um die vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Kompetenzzentren BBDC (Berlin Big Data Center, Leitung TU Berlin) und ScaDS (Competence Center for Scalable Data Services and Solutions, Leitung TU Dresden und Uni Leipzig) sowie das in Zusammenarbeit von Industrie und Forschung eingerichtete SDIL (Smart Data Innovation Lab, Leitung KIT). Diese drei Zentren werden zunächst in Kurzvorträgen vorgestellt. Eine sich anschließende Panel-Diskussion arbeitet Gemeinsamkeiten, spezifische Ansprüche und Kooperationsmöglichkeiten heraus.

2 Berlin Big Data Center (BBDC)

Im Rahmen des vom BMBF geförderten Kompetenzzentrums „Berlin Big Data Center“ (BBDC) unter der Federführung der Technischen Universität Berlin werden die Konsortialpartner Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Fritz-Haber-Institut der Max-Planck-Gesellschaft, Beuth Hochschule und Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI) in den Jahren 2014 bis 2018 neuartige und automatisch skalierbare Technologien entwickeln, welche tiefgreifende Analysen von „Big Data“ ermöglichen. Das BBDC verfolgt das Ziel, die Forschungsgebiete des Maschinellen Lernens und des Datenmanagements zu verschmelzen und ein hochskalierbares Open Source-System zu entwickeln, welches in der Lage ist, die Spezifikation, automatische Optimierung, Parallelisierung und Hardwareadaption sowie die fehlertolerante, effiziente Ausführung verschiedener Datenanalysemethoden (z.B. des Maschinellen Lernens, der linearen Algebra, der Statistik, der Wahrscheinlichkeitstheorie, der Computerlinguistik sowie der Signalverarbeitung) durchzuführen. Als Beispielanwendungen werden die Bereiche Gesundheit, Materialforschung, Informationsmarktplätze, Logistik und Industrie 4.0 zu betrachtet. Neben Forschung und Innovation werden im Kontext des BBDC auch mehrere Aktivitäten der Lehre gebündelt.

Volker Markl leitet das Fachgebiet Datenbanksysteme und Informationsmanagement (DIMA) an der Technischen Universität Berlin. Gleichzeitig ist Dr. Markl Status-Professor an der Universität von Toronto und leitet die Forschungsgruppe „Intelligente Analyse von Massendaten Smart Data“ am Deutschen Forschungszentrum für künstliche Intelligenz (DFKI). Seine Forschungsinteressen beinhalten neue Hardwarearchitekturen für das Informationsmanagement (z.B. neue Speichertechnologien wie NUMA, remote Memory oder Flash/Phase Change Memory, neue Verarbeitungstechnologien wie SIMD, Manycore), skalierbare Datenverarbeitung (Programmiermodelle, Anfrageübersetzung, -optimierung und -parallelisierung sowie skalierbares Data Mining) sowie Anwendungen in den Bereichen Text Mining und Informationsmarktplätze. Volker Markl ist auch Sprecher des „Data Analytics & Cloud“ Labs der TU Berlin und Direktor des vom BMBF eingerichteten Big Data Centers Berlin.

3 Competence Center for Scalable Data Services and Solutions (ScaDS)

Im Big-Data-Zentrum ScaDS (Competence Center for Scalable Data Services and Solutions) Dresden/Leipzig (www.scads.de) entwickeln international führende Forscher der TU Dresden und der Universität Leipzig neue Lösungen für datenintensive wissenschaftliche und wirtschaftliche Anwendungen. Profilbestimmende Forschungsschwerpunkte liegen in den Gebieten der Datenintegration, der Wissensextraktion sowie der visuellen Analyse. Aufgrund der erstklassigen Datenbankexpertise an beiden Standorten werden Techniken des Datenmanagement und der Datenanalyse umfassend vorangetrieben.

Erhard Rahm promovierte und habilitierte an der TU Kaiserslautern und leitet seit 1994 den Lehrstuhl für Datenbanken an der Universität Leipzig. Seit 2014 ist er wissenschaftlicher Ko-Koordinator des BMBF-geförderten Big-Data-Zentrums ScaDS (Competence Center for Scalable Data Services and Solutions) Dresden/Leipzig. Seine Forschungsinteressen liegen vor allem in den Gebieten der Datenintegration und Big Data. Er ist Autor von mehreren Büchern und über 200 wissenschaftlichen Publikationen. Seine Forschungsergebnisse im Gebiet der Datenintegration wurden bereits mehrfach international ausgezeichnet, u.a. durch zwei renommierte Test-of-Time Awards: den VLDB 10-Year Best Paper Award sowie den ICDE Influential Paper Award.

Wolfgang Lehner ist Direktor des Instituts für Systemarchitektur an der Fakultät Informatik der TU Dresden. Seine wissenschaftlichen Schwerpunkte liegen im Bereich der analytischen Auswertung großer Datenbestände, wobei stets versucht wird, eine Durchgängigkeit von der statisch-analytischen Modellierung bis zur effizienten Umsetzung auf Ebene der Systemarchitektur zu erreichen. Über 200 Veröffentlichungen und eine Vielzahl von Aktivitäten in der Community zeugen von einer regen wissenschaftlichen Aktivität und internationaler Vernetzung. Wolfgang Lehner vertritt als Fachkollegiat aktuell die Datenbank- und Informationssysteme innerhalb der DFG, ist Mitglied des VLDB Endowments und wurde 2014 in die Academy of Europe berufen. An der TU Dresden ist er neben dem Kompetenzzentrum ScaDS auch im SfB „HAEC – Highly Adaptive Energy Efficient Computing“, im Exzellenz-Cluster „cfAED – Advancing Electronics Dresden“ und als Sprecher im Graduiertenkolleg „RoSI – Rollenbasierte Informationssysteme“ aktiv.

4 Smart Data Innovation Lab (SDIL)

Das Smart Data Innovation Lab dient der Förderung der Smart Data Spitzenforschung in Deutschland. Durch Strukturierung von Big Data entstehen Informationen in Form von Smart Data, die zu Wissensvorteilen und zur Entscheidungsunterstützung genutzt werden können. Im SDIL werden in enger Zusammenarbeit von Industrie und Wissenschaft verbesserte Voraussetzungen für eine Spitzenforschung im Bereich Data Engineering/Smart Data geschaffen.

Michael Beigl ist seit 2010 Professor (W3) für Pervasive Computing Systems und Leiter des TECO am Karlsruher Institut für Technologie (TECO). Seit 2012 ist er darüber hinaus Dekan der Fakultät für Informatik. Zuvor hatte er von 2006 bis 2010 den Lehrstuhl für Verteilte und Ubiquitäre Systeme an der Technischen Universität Braunschweig inne und war 2005 Gastprofessor an der Keio Universität in Japan. Von 2000 bis 2005 leitete er als wissenschaftlicher Direktor des TECO der Universität Karlsruhe, ein Forschungslabor für anwendungsorientierte Forschung. Herr Beigl absolvierte sein Studium der Informatik an der Universität Karlsruhe und promovierte dort zum Dr.-Ing.. Er hat über 100 Artikel im Themenfeld des Mobile, Wearable, Ubiquitous und Pervasive Computing, des Internet der Dinge sowie der Datenanalyse veröffentlicht. Zu diesen Themen ist er in zahlreichen Gremien und Editorial Boards tätig und hat eine Vielzahl von Forschungsprojekten insbesondere zusammen mit der Industrie durchgeführt sowie Start-Ups gegründet. Seit 2014 ist Herr Beigl Sprecher des Smart Data Innnovation Lab (SDIL).

Dissertationspreise

Forecasting in Database Systems

Ulrike Fischer

Database Technology Group
Technische Universität Dresden
Nöthnitzer Str. 46
01187 Dresden
ulrike.fischer@tu-dresden.de

Abstract: Time series forecasting is crucial in a number of domains such as production planning and energy load balancing. In these areas, forecasts are often required by non-expert users on large multi-dimensional data sets expecting short response times. However, as current traditional database systems support forecasting only in a limited and non-declarative way, it is performed outside the database system by specially trained experts. We introduce a novel approach that seamlessly integrates time series forecasting into an existing database management system. In contrast to flash-back queries that allow a view on the data in the past, we have developed a Flash-Forward Database System (F^2DB) that provides a view on the data in the future. It supports a new query type — a forecast query — that enables forecasting of time series data for any user and is automatically processed by the core engine of an existing DBMS. We introduce various optimization techniques for three different types of forecast queries: ad-hoc queries, recurring queries, and continuous queries. All approaches intend to increase the efficiency of forecast queries while ensuring high forecast accuracy.

1 Introduction

Time series forecasting has been subject to intensive research for a long period of time and still offers many open research problems [GH06]. This comes as no surprise as forecasting is crucial for a number of application areas. The Data Warehousing Institute (TDWI) surveyed about 400 IT professionals and data analysts, where nearly 80% of the respondents said "reacting more quickly to changing market conditions is a business benefit they seek" [Sto12]. The overall business goals are often to use predictive insights to anticipate, rather than react to customer behavior.

In the past, time series forecasting was performed by highly qualified statistical experts, with long experience in the company, who manually experimented with different forecasting algorithms and parametrizations using dedicated statistical software environments. Over the last few years, we are experiencing a paradigm shift, which is not just related to forecasting but valid for any kind of sophisticated statistical algorithm. We can observe the transition of traditional database systems to big data stores where massive amount of data arrives continuously in real-time from vast data sources. This development results not only in big challenges but also big opportunities for analyzing large data sets in real-time,

which further leads to better-informed business decisions. As a consequence, modern data analysis in database management systems involves increasingly sophisticated statistical methods that go well beyond the roll-up and drill-down over simple aggregates of traditional BI [CDD+09]. Moreover, sophisticated models and advances in autonomic model selection and self-management make advanced data analytics more useful for users and applications. On the one hand, data mining is increasingly performed by people who are not mathematicians or statistical experts. Many users do not want to concern themselves with tasks like model selection and training and are satisfied with "good enough" forecasts [ACR+11]. On the other hand, there is a frequent need for fully automatic forecasting without any human intervention [HKSG02], e.g., balancing energy demand and supply.

Several research groups (e.g., [CDD+09], [KNR13]) and major database vendors (e.g., [MYC05], [GLW+11]) are making an effort towards integrating sophisticated statistical algorithms, such as time series forecasting, into classical data management platforms. However, many existing approaches follow a black-box style and try to keep changes to the database system as minimal as possible, for example, by using user-defined functions or by connecting the database system with a statistical software environment. While such approaches are more general and easier to realize, they miss significant opportunities for performance improvements. Moreover, most methods still approach the problem with statistical models and statistical algorithms, and try to get a database system to act like a statistical software environment. Such approaches ignore the concept of declarative database queries that do not care about the actual computation of query results.

In the thesis [Fis14], we introduce a novel approach that seamlessly integrates time series forecasting into an existing database management system. A tight coupling of the time series forecasting process with a DBMS ensures consistency between data and models and increases the usability as well as the efficiency of the forecasting process. In contrast to flash back queries that allow a view on the data in the past, we have developed a *Flash-Forward Database System* (F^{2}DB) that provides a view on the data in the future. It supports a new query type — a forecast query — that enables forecasting of time series data and is automatically and natively processed by the core engine of an existing DBMS.

In the following brief summary of the thesis, Section 2 gives a background on time series forecasting. Section 3 introduces the conceptional architecture of our flash-forward database system. Sections 4–6 highlight the different components and optimization techniques of F^{2}DB. Finally, we conclude in Section 7.

2 Foundations of Time Series Forecasting

Time series forecasting is best described by a generalized model-based forecasting process, from which we can derive typical requirements on a forecasting system as well as realization possibilities inside a DBMS.

Model-Based Forecasting A *time series* is a sequence of observations taken sequentially in time, spaced at equidistant time intervals. *Forecasting* refers to the estimation of values of a time series at future points in time, so called *forecast values* or short *forecasts*.

Hereby, the *forecast horizon* describes the interval from the next point in time up to a given future point in time. A *forecasting method* is a procedure for computing forecasts from present and past values. Most forecasting methods are based on a *forecast model*, which is learned over historical training data and used to compute forecast values (e.g., Exponential Smoothing or ARIMA models [BJR08]). *Model-based time series forecasting* consists of three essential steps. First, a forecast model is created by defining input, output as well as the forecasting method, and by estimating the model parameters (*model creation*). Second, forecast values based on the created forecast model are calculated (*model usage*). Last, the forecast model is evaluated by comparing real time series data with forecast values and, optionally, model adaption is triggered by recalculating the model parameters or choosing a new model (*model maintenance*). In model-based forecasting, the time-consuming part poses the model creation step, whereas model usage only requires the application of a function with the trained parameters.

System Requirements Time series forecasting is an important technique for various application areas such as production planning [MB98], energy balancing, and online display advertisement. A system that supports the introduced model-based forecasting process as well as typical application domains has to follow certain requirements. First, we require a suite of widely used forecasting techniques applicable for different areas. Furthermore, a generic interface would be desirable that enables the inclusion of new domain-specific forecasting methods. A simple and declarative query language allows the application of forecasting by any users without statistical expertise (e.g., supply chain managers). Real-time requirements (e.g, a few hundred milliseconds in display advertisement) over large multi-dimensional data sets demand the efficient creation of forecast models. However, as model creation is expensive, mechanisms have to be provided that allow the storage and reuse of forecast models for different queries and users. Consequently, forecast models have to be continuously and efficiently adapted to changes in the time series behavior.

Architectural Integration Numerous works have addressed the integration of analytical methods inside a DBMS. Existing methods can be classified into (1) no-database integration approaches that use external software (Matlab, R), (2) partial integration approaches that try to keep changes to the database as small as possible (e.g., [CDD+09, GLW+11]), and (3) full integration approaches that actually extend the functionality of a database system (e.g., [DM06, ACR+11]). External systems often contain a large number of statistical forecasting methods, but lack database features such as declarative forecast queries, query optimization, or model storage and reuse. Partial integration approaches take a first step towards the integration of the forecasting lifecycle within a DBMS and integrate forecasting into relational query processing to some extent. They, however, still treat the forecasting process as a black-box approach, either within a user-defined, a customized function, or within a R script. Subsequently, all decisions have to be made locally within the specific function, allowing no joint model reuse, no automatic maintenance, or multi-dimensional physical design. In contrast, models are handled as first class citizens and automatically maintained by some of the full integration approaches, such as MauveDB [DM06]. However, existing approaches do not support declarative forecast queries as they require the explicit selection of a model in a query. Furthermore, most approaches do not focus on time series forecasting techniques and optimizations.

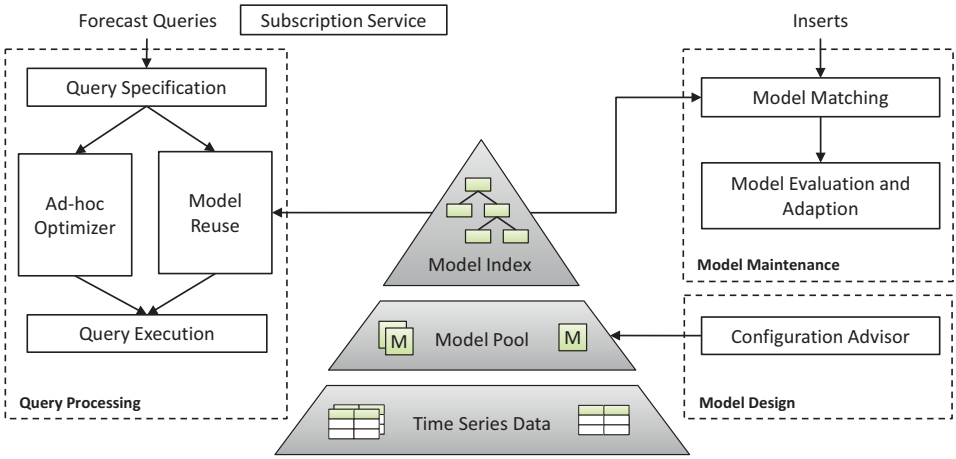


Figure 1: Flash-Forward Database System F^{2}DB

3 A Flash-Forward Database System

Based on the discussed system requirements and the limitations of existing work, in this section, we propose a general architecture that integrates the whole forecasting lifecycle natively into a database management system. In contrast to flash-back queries that allow a view on the data in the past, we propose a *Flash-Foreward Database System* (F^{2}DB) that provides a view on the data in the future. Figure 1 shows the main conceptional components of F^{2}DB . Based on this architecture, a prototype was developed within the open-source DBMS PostgreSQL [FRL12a].

Analogue to traditional database views, a *time series* can be specified by arbitrary database queries and represented as a special time series view. *Forecast models* on time series are stored as first-class citizens in a common *model pool*. Furthermore, models are indexed in a specialized index structure that efficiently finds existing models for a given forecast query and insert operation (*model index*).

The processing of declarative forecast queries requires the extension of the traditional query processing engine. *Query specification* recognizes new forecast-specific keywords and further analyzes the given query. Hereby, we can either search and *reuse* existing models from the model index or create new models at query runtime. The latter is supported by the *ad-hoc optimizer*, which includes various optimization techniques to increase query runtime as well as query accuracy. Finally, the query is *executed* by new forecast-specific physical operators. Insert operations, containing new time series tuples, require the maintenance of models in the model pool. Hereby, *model matching* finds existing models that are based on those inserts. Models are then *evaluated* and *adapted* if necessary. Forecast model maintenance poses additional overhead to the database system. In the style of traditional materialized view and index advisors, the *configuration advisor* suggests a physical design of forecast models for a given query workload. Hereby, the advisor is faced with

the tradeoff of reducing model maintenance costs and increasing forecast query efficiency and accuracy. A last component, the *subscription service*, allows applications to register a query once at F^{2}DB. As new time series values arrive, it automatically sends notifications to the application based on time and accuracy constraints.

4 Processing Forecast Queries

In this section, we discuss the basics of forecast query processing and propose optimization techniques for ad-hoc forecast queries.

Forecast Queries in SQL Support for forecasting requires the extension of the SQL data manipulation language with forecast-specific keywords. In the SQL:2011 standard, a flashback query retrieves data as it existed at some point in the past using the `AS OF` clause. We use the same language construct to retrieve data as it will exist in the future. For example, the following forecast query requests sales forecasts of mobile phones:

```
SELECT    orderdate, quantity
FROM      facts
WHERE     pgroup = 'phones'
GROUP BY  orderdate
AS OF     now() + interval '3 months'
```

Our goal is to keep the query language as minimal as possible, for which reason only the `AS OF` keyword is mandatory. However, we offer additional language constructs to specify information about input and output variables, or the forecasting method [FRL12a].

Logical Forecast Operator Forecasting introduces a novel semantic that is not covered by existing database operators. Based on existing input tuples, representing a historical time series, new approximate tuples are outputted, representing the future. This leads to a novel logical operator, the *forecast operator*, and interesting interaction schemes with existing operators. Most importantly, as the output of the forecast operator is only approximate, we have to extend the traditional semantic of equivalence of query plans. In the thesis, we analyze the relationship of the forecast operator with traditional database operators and explore possibilities to create a forecast model within a query plan.

Execution Model and Physical Plan Operators Analogue to traditional operators, the forecast operator can be mapped to different physical implementations, covering the variety of available forecasting methods, including general techniques as well as special forecasting methods for different domains. Forecasting methods are captured by a black-box-style generic internal *forecasting method interface* (FMI) that allows the addition of new methods by implementing predefined functions and registering them in the system catalog. Such a black-box approach is not restricted to a particular class of forecasting methods and allows the reuse of existing forecasting libraries behind the simple interface. Following the general forecasting process, the interface decouples model creation and model usage functionality and is exploited by two new physical plan operators. The `CreateModel` operator is responsible for model creation. It receives a time series rela-

tion as input and outputs a set of forecast models. Subsequently, the `Forecast` operator receives as input a set of forecast models and outputs a time series relation containing corresponding forecast values. To include the new operators into query optimization, we have developed novel cost models.

Sample-Based Query Optimization In contrast to traditional query optimization, forecast queries are approximate per se. We can reduce the amount of input data to the forecast operator and still compute valid results with a similar accuracy. Sampling based methods reduce the amount of processed data and have been successfully applied in many scenarios. In our case, we apply sampling in two different manners. *Vertical sampling* reduces the number of processed time series in a query [FRL12b], whereas *horizontal sampling* reduces the length of the time series. Both approaches reduce the processed data and subsequently the query runtime without or little loss in forecast accuracy. Vertical sampling results in missing time series in a query, which have to be estimated and integrated into the overall query result. Hence, we need to focus on how to estimate these missing time series. In contrast to traditional sampling that deals with single values, we sample complete time series. We propose novel estimators and sample selection strategies that exploit this fact based on the time series history. Horizontal sampling only reduces the history length, but does not remove intermediate time series values. Subsequently, the forecast value computation is exactly the same, but might result in different accuracy. However, in contrast to vertical sampling, an increasing sample size does not necessarily increase the forecast accuracy. A longer history length might have no effect at all (depending on the forecasting method) or even lead to an accuracy decrease. We introduce an empirical optimization technique that evaluates several history lengths at runtime on a sample of time series.

Evaluation In our evaluation, we compare the end-to-end performance of our in-DBMS implementation to alternative integration approaches and evaluate the overall benefit of the proposed optimization techniques. For our end-to-end comparison, we adapt the classical TPCH benchmark to be able to process forecast queries, denoted as *F-TPCH*. As the TPCH benchmark is centered around a data-warehouse scenario with OLAP-style queries, it is well suited for advanced analytics such as time series forecasting. The adaption of the TPCH benchmark requires changes to the data generation as well as query generation components of the benchmark. Furthermore, to show the accuracy of our techniques, we acquired several real-world data sets from various domains. Our experimental evaluation shows that the integration of forecasting inside the database leads to significant speed ups compared to external forecasting. Query runtime and forecasting accuracy is further improved by applying sample-based optimization.

5 Materialization of Forecast Models

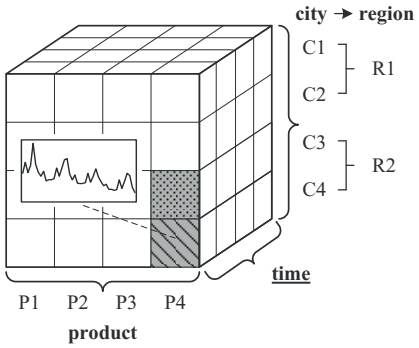
We now turn our attention from ad-hoc to recurring forecast queries. Specifically, our goal is to decrease the runtime and increase the accuracy of forecast queries by preselecting and materializing forecast models. Hereby, we want to retain the transparent processing of forecast queries from the last section, which requires a mechanism to transparently find existing models in the database.

Model Index Materialization of forecast models is realized by two new data structures. The *model pool* contains all currently existing forecast models. Each model is associated with static and dynamic meta data. A second data structure, the *model index*, identifies matching models for incoming query and insert transactions [FRBL10]. The index is kept as in-memory structure, as it needs to be accessed for every forecast query and insert transaction. The key of the model index is the input data of the model, whereas the model itself is the value. Hereby, a predicate index is responsible for finding predicates that satisfy certain query conditions or tuple values, which are further matched against signatures of *where expressions* to actually identify the affected models. This allows arbitrary forecast queries and enables expression sharing of different queries.

Forecast query processing can now take two different routes. On the one hand, if a model is available, forecast values are directly computed from this model without accessing the base data on disk. On the other hand, if no model is found in the model index, we create a new model as done in the previous section. In addition, we store the model in the model index so it can be reused by subsequent queries. As new time series values arrive, we need to identify and maintain all affected models in the model index. First of all, this involves the update of the internal model state to the current time series values, which is a fast and incremental operation. However, updates may also reflect changes in the time series behavior and these can only be incorporated by reestimating the model parameters. Since this step is computationally expensive, we propose several evaluation and adaption strategies to reduce the effort for full maintenance.

Optimization of Model Configurations Forecast model maintenance poses additional overhead to the database system. Materializing and maintaining a model for each single time series is infeasible for large multi-dimensional data sets as found in typical data warehouse environments. Existing forecasting literature already provides some initial techniques to reuse models, for example, by using low-level models to compute forecasts for aggregated time series data. Interestingly, those techniques do not only reduce the number of models in a physical design, they can even increase the accuracy of the forecast result.

Consider the example of forecasting sales data with sales quantity as measure and time, product, city, and region as dimensional attributes (Figure 2). The dimension time together with the measure forms time series for different products and locations. Forecast Query 1 in Figure 2 demands the future of the *base time series* representing the dashed area, i.e., sales forecasts of product $P4$ in city $C4$ over the next day. Forecast Query 2 (dashed and dotted area) requests forecasts of product $P4$ in region $R2$ (containing cities $C3$ and $C4$) and, thus, is an example of forecasting an *aggregated time series*. To answer Forecast Query 2 of Figure 2 we may exploit various approaches. Most current state of the art would create a forecast model over the aggregated time series (dashed and dotted area). However, alternatively, we might create two forecast models, one over city $C3$ and one over city $C4$, and calculate the forecast of Query 2 by aggregating the individual forecasts. Third, we could create a model over all cities ($C1$-$C4$) and then use disaggregation to just compute the forecasts of the cities in region $R2$. Finally, we might also exploit similar time series (e.g., sales of product $P3$ in region $R2$) and derive the forecasts from a model created over this time series. Each approach leads to a different forecast accuracy and different costs as we have to maintain all models stored in the database.



Query 1 (dashed area)

```
SELECT    time, sales
FROM      facts
WHERE     product = 'P4'
AND       city = 'C4'
AS OF     now() + interval '1 day'
```

Query 2 (dashed + dotted area)

```
SELECT    time, SUM(sales)
FROM      facts
WHERE     product = 'P4'
AND       region = 'R2'
GROUP BY  time
AS OF     now() + interval '1 day'
```

Figure 2: Forecast Queries in Multi-Dimensional Data Sets

Model Configuration Advisor In the style of traditional materialized view and index advisors, we introduce a forecast model advisor that suggests a physical design of forecast models for a given query workload [FSHL13]. The technical challenge of the advisor is the fact that there are no known ways to estimate the accuracy of a physical design of forecast models without actually deploying and querying it. However, deploying a physical design is costly since statistical models can take a long time to be built, especially for a large number of time series.

The model configuration advisor receives a data set and associated query workload as input. It outputs a *model configuration* as well as associated *forecast error* and *model costs*. The advisor is based on an iterative process that selects a set of candidate time series in each iteration for which a model should be built and analyzed. This iterative process ensures that the advisor can be canceled at any time, if either (1) the forecast error is acceptable or only shows slight improvements with more models or (2) the maximum acceptable model costs are reached. To find model candidates, we use *indicators* that heuristically indicate the expected benefit of a model without removing or, more importantly, building it. Then, in the evaluation phase, the benefit of adding or deleting candidate models is empirically evaluated by explicitly creating forecast models and executing the given query workload. Parameters of the advisor like the number of candidate models are automatically tuned in a control phase. Our evaluation shows that our advisor is able to achieve a higher forecast accuracy than alternative approaches and that we are able to calculate a configuration in reasonable time even for larger data sets and complex models.

6 Subscription-Based Forecast Queries

We finally extend forecast queries with continuous aspects, allowing an application (subscriber) to register a query once at F^{2}DB. We call such kinds of queries *subscription-based forecast queries* [FBLP12]. Subscription-based forecast queries arise when an application continuously requires forecast values for further processing. Forecast queries contain the

unique characteristic that they can provide an arbitrary number of forecast values, i.e., by adjusting the forecast horizon. However, with each new actual value the underlying model is updated and better forecasts might be available. A dependent application could obtain these values by repeatedly polling from the database. This is very inefficient if forecasts have changed only marginally, especially if the application executes a computational expensive algorithm based on the received forecasts. The subscriber therefore wishes to be notified only when forecast values have changed relevant to the application.

A subscription-based forecast query registers at the database system given various parameters, e.g., the time series to forecast, a continuous forecast horizon, and a threshold of acceptable forecast deviations. For example, a simple forecast query might request continuous forecasts of energy demand for the next 2 hours with a threshold of 10%. As time proceeds and new time series values arrive, models are updated and optionally maintained by the DBMS. This results in better forecasts leading to notifications sent to the subscriber that contain at least the forecast horizon specified by the subscriber (denoted as *notification length*). In the previous example, a notification needs to be sent if the subscriber holds less than 2 hours of forecast values or if new forecasts are available that deviate by more than 10% from old forecasts sent before. The subscriber processes all notifications, where the processing costs of the subscriber depend on the notification length, which further influences the number of notifications. These *subscriber costs* can be communicated to the database system to optimize future notifications.

Our objective is the reduction of the processing costs of the subscriber by trading the number of notifications against the notification length. We approach this challenge by introducing three computational approaches, which are based on the available historical time series data and the cost model of the subscriber. The first one intends to determine a static notification length. The second one determines different notification lengths according to different time slices. Finally, the third one determines the next notification length online. Our experimental evaluation shows the superiority of our computational approaches over alternatives, a significant reduction of the subscriber costs with low computational overhead as well as the validity of our cost model in real-world situations.

7 Conclusions

We proposed the deep integration of forecasting within a traditional database management system. Declarative forecast queries enable forecasting for any user, especially those not trained in statistics. Arbitrary forecast queries allow the processing of the forecast itself inside the DBMS and the association with other source data (e.g., through joins). Forecast models are handled as first-class citizens and transparently processed inside the DBMS. This allows the materialization and reuse of models by multiple queries as well as transparent query optimization. The introduced novel flash-forward database system opens up many interesting research directions. For example, future work could look at new forecast query optimization techniques, develop parallel forecast operators, or investigate the online maintenance of model configurations.

References

[ACR+11] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. The Case for Predictive Database Systems: Opportunities and Challenges. In *CIDR*, pages 167–174, 2011.

[BJR08] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis: Forecasting and Control, 4th ed.* Wiley, 2008.

[CDD+09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2:1481–1492, 2009.

[DM06] Amol Deshpande and Samuel Madden. MauveDB: Supporting Model-based User Views in Database Systems. In *SIGMOD*, pages 73–84, 2006.

[FBLP12] Ulrike Fischer, Matthias Boehm, Wolfgang Lehner, and Torben Bach Pedersen. Optimizing Notifications of Subscription-Based Forecast Queries. In *SSDBM*, pages 449–466, 2012.

[Fis14] Ulrike Fischer. *Forecasting in Database Systems*. PhD thesis, Technische Universität Dresden, 2014.

[FRBL10] Ulrike Fischer, Frank Rosenthal, Matthias Böhm, and Wolfgang Lehner. Indexing Forecast Models for Matching and Maintenance. In *IDEAS*, pages 26–31, 2010.

[FRL12a] Ulrike Fischer, Frank Rosenthal, and Wolfgang Lehner. F2DB: The Flash-Forward Database System. In *ICDE*, pages 1245–1248, 2012.

[FRL12b] Ulrike Fischer, Frank Rosenthal, and Wolfgang Lehner. Sample-Based Forecasting Exploiting Hierarchical Time Series. In *IDEAS*, pages 120–129, 2012.

[FSHL13] Ulrike Fischer, Christopher Schildt, Claudio Hartmann, and Wolfgang Lehner. Forecasting the Data Cube: A Model Configuration Advisor for Multi-Dimensional Data Sets. In *ICDE*, 2013.

[GH06] Jan G. De Gooijera and Rob J. Hyndman. 25 Years of Time Series Forecasting. *International Journal of Forecasting*, 22:443–473, 2006.

[GLW+11] Philipp Große, Wolfgang Lehner, Thomas Weichert, Franz Färber, and Wen-Syan Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *PVLDB*, 4:1307–1317, 2011.

[HKSG02] Rob J. Hyndman, Anne B. Koehler, Ralph D. Snyder, and Simone Grose. A State Space Framework for Automatic Forecasting Using Exponential Smoothing Methods. *International Journal of Forecasting*, 18:439–454, 2002.

[KNR13] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: Making it Easier to Build and Maintain Big-data Analytics. *Queue*, 11(1):30–346, 2013.

[MB98] John T. Mentzer and Carol C. Bienstock. The Seven Principles of Sales-Forecasting Systems. *Supply Chain Management Review*, pages 76–83, 1998.

[MYC05] Boriana L. Milenova, Joseph S. Yarmus, and Marcos M. Campos. SVM in Oracle Database 10g: Removing the Barriers to Widespread Adoption of Support Vector Machines. In *PVLDB*, pages 1152–1163, 2005.

[Sto12] David Stodder. Customer Analytics in the Age of Social Media. The Data Warehousing Institute Research, July 2012.

An Overview on Querying and Learning in Temporal Probabilistic Databases

Maximilian Dylla*

Google
Paris, France
maxdylla@google.com

Abstract: Probabilistic databases store, query and manage large amounts of uncertain information in an efficient way. This paper summarizes my thesis which advances the state-of-the-art in probabilistic databases in three different ways: First, we present a closed and complete data model for temporal probabilistic databases. Queries are posed via temporal deduction rules which induce lineage formulas capturing both time and uncertainty. Second, we devise a methodology for computing the top-k most probable query answers. It is based on first-order lineage formulas representing sets of answer candidates. Moreover, we derive probability bounds on these formulas which enable pruning low-probability answers. Third, we introduce the problem of learning tuple probabilities, which allows updating and cleaning of probabilistic databases, and study its complexity and characterize its solutions.

1 Introduction

In the last decade, we saw another big rise in the amount of digital information. Database systems play a key role in managing this data because of their abilities in storing, querying, and updating large data collections. One of the basic assumptions in database systems is that the data is certain: a data record is either a perfect implementation of some real-world truth or, if it does not hold, absent in the database. In reality, however, a large amount of data is not deterministic, but rather uncertain. For example, coarse-grained input can result in uncertainty. For instance, a sensor with limited precision is inherently uncertain with respect to the precise physical value. Moreover, ambiguity can cause uncertainty. For instance, most sentences in natural language allow more than one interpretation, sometimes even leaving the reader in doubt. One way to store, query and manage large amounts of uncertain data are probabilistic databases (PDBs) [SOCK11]. My thesis [Dyl14] advances the field of probabilistic databases in several ways, where this paper serves as a brief overview.

Contributions and Outline. In Section 2, we devise a *temporal-probabilistic data model* which supports both time and probabilities as first-class citizens. In Section 3, we present

*This research was conducted while attending Max Planck Institute for Informatics. The views and conclusions contained herein are those of the author and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of Google.

an approach for computing the *top-k query answers*, which feature the highest probabilities among all answers. Our main technical tool are *first-order lineage* formulas, which can represent both entire sets of query answers and partially evaluated grounding states. We formally derive *probability bounds* for these formulas, enframing the probabilities of all answers represented by the formulas. In Section 4, we establish a methodology to *learn tuple probabilities* from labeled lineage formulas. We characterize the problem theoretically by investigating its *complexity* as well as the *nature of its solutions*.

2 Temporal Probabilistic Databases

In recent years, temporal databases [Jen00] (where tuples are valid only at some points in time) and probabilistic databases [SOCK11] (whose tuples exist with a probability only) have emerged as two intensively studied areas of database research. So far, the two fields have however been investigated largely only in isolation. In this section, we sketch a temporal-probabilistic database (TPDB) model [DMT13, Dyl14] supporting both temporal as well as probabilistic data.

Example 1. *Our running example is based on information extraction which aims to automatically harvest factual knowledge from web sources. The temporal probabilistic database of Figure 1 captures a number of extracted facts around the actor "Robert DeNiro" and their origin, that is, the web domain they were extracted from. The probability of each tuple represents to which degree we believe this extraction is correct. Tuple I_1 expresses that* DeNiro *got married to* Abbott *on April 28th, 1976, which is encoded into the time interval* $[1976\text{-}04\text{-}28, 1976\text{-}04\text{-}29)$. *The time and probability annotations together express that this tuple is* true *for the given time interval with probability* 0.2*, and it is* false *for this interval with probability* 0.8. *Note that we allow relations without temporal annotations, e.g.,* FromDomain*, whose tuples are valid at all time points.*

WeddingExtraction

	Person$_1$	Person$_2$	Did	Valid Time	p
I_1	*DeNiro*	*Abbott*	1	$[1976\text{-}04\text{-}28, 1976\text{-}04\text{-}29)$	0.2
I_2	*DeNiro*	*Abbott*	2	$[1976\text{-}04\text{-}28, 1976\text{-}04\text{-}29)$	0.9
I_3	*DeNiro*	*Hightower*	2	$[1997\text{-}01\text{-}01, 1998\text{-}01\text{-}01)$	0.6

FromDomain

	Did	Domain	p
I_5	1	*Wikipedia.org*	1.0
I_6	2	*People.com*	1.0

DivorceExtraction

	Person$_1$	Person$_2$	Did	Valid Time	p
I_4	*DeNiro*	*Abbott*	1	$[1988\text{-}09\text{-}01, 1988\text{-}12\text{-}01)$	0.4

Figure 1: An Example Temporal Probabilistic Database

Time. As a fist step, we introduce our notion of time. We consider the *time universe* $\mathcal{U}^T$ as a linearly ordered, finite sequence of *time points*, e.g., days, minutes or even milliseconds. Then, a *time-interval* consists of a contiguous and finite set of time points over $\mathcal{U}^T$, which we denote by a half-open interval $[t_b, t_e)$ where $t_b, t_e \in \mathcal{U}^T$ and t_b before t_e.

Example 2. *Regarding Figure 1 we can choose $\mathcal{U}^T$ to be the sequence of all days of the 20th century.*

Temporal Relation. Before we formally define TPDBs, we establish a useful property of temporal relations, e.g., *WeddingExtraction*. Assume we drop *Did* as an argument of *WeddingExtraction*, then a probabilistic database engine might conclude that *DeNiro* got married twice on April 28th, 1976 with different probabilities. To resolve this issue, we enforce the time-intervals of tuples with identical non-temporal arguments $\bar{a}$ (e.g., the first three columns of *WeddingExtraction*) to be disjoint, termed *duplicate-free* [DBG12].

Definition 1. *A temporal relation instance $\mathcal{R}$ is called* duplicate-free*, if for all pairs of tuples $R(\bar{a}, t_b, t_e), R(\bar{a}', t'_b, t'_e) \in \mathcal{R}$ it holds that:*

$$\bar{a} = \bar{a}' \;\Rightarrow\; [t_b, t_e) \cap [t'_b, t'_e) = \emptyset$$

The above definition is fulfilled for all relation instances of Figure 1, since I_1, I_2, and I_3 have differing non-temporal arguments.

Temporal Probabilistic Database. Now, we define a *temporal probabilistic database* as the triple $(\mathcal{T}, p, \mathcal{U}^T)$ [DMT13, Dyl14]. Here, $\mathcal{T}$ is the set of all tuples across all relations (which we assume to be duplicate-free). Furthermore, p is a function $p : \mathcal{T} \to (0, 1]$ which assigns a non-zero probability value to each tuple. The probability values of different tuples are assumed to be independent. Finally, $\mathcal{U}^T$ is the time universe as defined before.

Example 3. *For the database of Figure 1 we have $\mathcal{T} = \{I_1, \ldots, I_9\}$ and $p(I_1) = 0.2$.*

Temporal Deduction Rules. To derive new knowledge from a TPDB, we employ temporal deduction rules. These can be viewed as generally applicable "if-then-rules". Formally, deduction rules have the shape of a logical implication with a conjunction of both positive and negative literals in the body and exactly one positive literal in its head. This class of rules coincides with safe Datalog without recursion [AHV95] or nested select-project-join queries in SQL.

Example 4. *Given the tuples of Figure 1 we first aim to reconcile the facts, i.e., when* DeNiro *got married and divorced, by writing these two temporal deduction rules:*

$$\underbrace{Wedding(P_1, P_2, T_b, T_e)}_{head} \leftarrow \underbrace{\begin{pmatrix} WeddingExtraction(P_1, P_2, Did, T_b, T_e) \\ \wedge\ FromDomain(Did, D) \end{pmatrix}}_{body} \quad (1)$$

$$Divorce(P_1, P_2, T_b, T_e) \leftarrow \begin{pmatrix} DivorceExtraction(P_1, P_2, Did, T_b, T_e) \\ \wedge\ FromDomain(Did, D) \end{pmatrix} \quad (2)$$

Both rules propagate the persons P_1 and P_2 and the time interval as specified by $[T_b, T_e)$ to the head relation, e.g., Wedding *or* Divorce*. More interestingly, we are now able to deduce the time interval of their marriage starting at the beginning of the wedding and ending at the end of the divorce:*

$$Marriage(P_1, P_2, T_{b,1}, T_{e,2}) \leftarrow \begin{pmatrix} Wedding(P_1, P_2, T_{b,1}, T_{e,1}) \wedge \\ Divorce(P_1, P_2, T_{b,2}, T_{e,2}) \ \wedge \end{pmatrix} T_{e,1} <^T T_{b,2} \quad (3)$$

Thereby, we consider only weddings that took place before divorces as stated by the condition $T_{e,1} <^T T_{b,2}$. If a couple is still married, we let the time interval of their marriage

start at the wedding until the last possible time point (denoted by the constant t_{max}).

$$Marriage(P_1, P_2, T_b, t_{max}) \leftarrow (Wedding(P_1, P_2, T_b, T_e) \ \wedge \ \neg Divorce'(P_1, P_2)) \quad (4)$$

Here, the existence of any divorce independent of time is modeled by the projection:

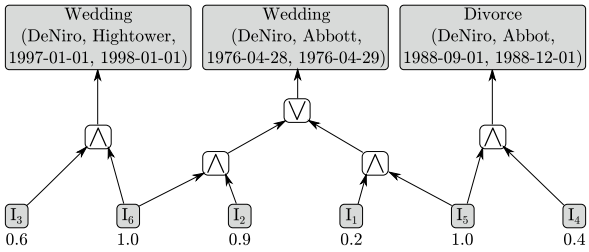
$$Divorce'(P_1, P_2) \leftarrow Divorce(P_1, P_2, T_b, T_e)$$

Queries. Based on deduction rules we define a query to be a conjunction of head literals.

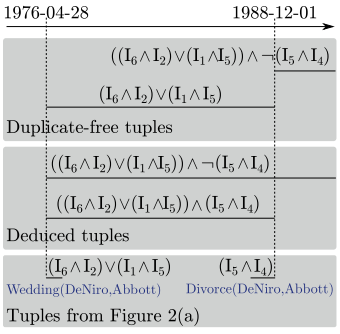
Example 5. *Extending Example 4, we formulate the query $Wedding(P_1, P_2, T_1, T_2) \wedge Divorce(P_1, P_2, T_2, T_3)$, which asks for persons P_1, P_2 who got divorced right after their wedding.*

Lineage. Next, we instantiate the deduction rules over the TPDB, i.e., we employ the rule to deduce new tuples, which is called grounding. Thereby we trace the deduction history of the new tuples. In database terminology this is called *data lineage* [BDSH+08], which we represent by a propositional formula. More detailed, lineage relates each new tuple with the tuples $\mathcal{T}$ via the three Boolean connectives $\wedge$, $\vee$ and $\neg$. These reflect the semantics of the relational operations that were applied to deduce that tuple.

Example 6. *If we apply the rules of Equations* (10) *and* (11) *to the tuples of Figure 1, we obtain the lineage formulas depicted in Figure 2(a). There, Equation* (10) *can be instantiated twice to deduce* Wedding(DeNiro,Abbott,1976-04-28,1976-04-29)*, which results in the two conjunctions in the middle. Then, both conjunctions are connected by a disjunction representing these two options.*



(a) Lineage Formulas of Example 6



(b) Lineage and Deduplication

Figure 2: Lineage

In general, if we execute temporal deduction rules, the newly deduced tuples may not necessarily define a duplicate-free relation. We illustrate this issue by the following example.

Example 7. *In Example 6, we deduced tuples about DeNiro's wedding to and divorce from Abbott which are displayed on the bottom of Figure 2(b). Applying the deduction rules of Equations* (3) *and* (4) *to these tuples yields the tuples in the middle of the figure, which have equivalent non-temporal arguments, i.e.,* DeNiro *and* Abbott*, but their time-intervals are overlapping, which contradicts Definition 1 of duplicate-free relation instances.*

Hence, in order to convert a temporal relation instance with duplicates (as shown in the middle of Figure 2(b)) into a duplicate-free temporal relation (as shown on the top of Figure 2(b)), we provide the following definition [DMT13, Dyl14]. Let $\lambda(I)$ stand for the lineage formula attached to a deduced tuple I.

Definition 2. *Let a temporal relation R, non-temporal constants $\bar{a}$, a time point $t \in \mathcal{U}^T$, and a set of tuples $\mathcal{T}$ be given. Then, L is defined as the set of lineages of tuples $R(\bar{a}, t_b, t_e)$ that are valid at time point t:*

$$L(R, \bar{a}, t, \mathcal{T}) := \{\lambda(I) \mid I = R(\bar{a}, t_b, t_e) \in \mathcal{T}, t_b \leq t < t_e\}$$

We create duplicate free tuples $I' = R(\bar{a}, t_b, t_e)$ *such that for any pair of time points $t_0, t_1 \in [t_b, t_e)$ it holds that:*

$$L(R, \bar{a}, t_0, \mathcal{T}) = L(R, \bar{a}, t_1, \mathcal{T}) \tag{5}$$

Furthermore, we define the new tuples' lineages to be:

$$\lambda(I') := \bigvee_{\phi_i \in L(R, \bar{a}, t_b, \mathcal{T})} \phi_i \tag{6}$$

In short, for each time-point t we create the disjunction of all tuples being valid at t (see Equation (6)). More detailed, for a given relation instance and the non-temporal arguments of a tuple, L is the set of all tuples' lineages that share the same non-temporal arguments and which are valid at time point t. Hence, consecutive time-points for which L contains the same lineage formulas form the new intervals (see Equation (5)).

Example 8. *Applying Definition 2 to the tuples shown in the middle of Figure 2(b) yields the tuples shown at the top of the figure. For instance, if we inspect L at the time points* 1988-11-30 *and* 1988-12-01*, we notice that $\{((I_6 \wedge I_2) \vee (I_1 \wedge I_5)) \wedge (I_5 \wedge I_4), ((I_6 \wedge I_2) \vee (I_1 \wedge I_5)) \wedge \neg(I_5 \wedge I_4)\} \neq \{((I_6 \wedge I_2) \vee (I_1 \wedge I_5)) \wedge \neg(I_5 \wedge I_4)\}$, so two different tuples have to be kept in the relation.*

Probability Computations. Since in a probabilistic database, each database tuple exists only with a given probability, we can quantify the probability that each deduced tuple exists. Formally, we compute the *probability* $P(\phi)$ of any lineage formula ϕ over tuples in $\mathcal{T}$ as the sum of the probabilities of all subsets $\mathcal{W} \subseteq \mathcal{T}$. For this, we consider only subsets, which satisfy ϕ, i.e., $\mathcal{W} \models \phi$, by setting all tuples in $\mathcal{W}$ to be *true* and all others to be *false* [SOCK11]:

$$P(\phi) := \sum_{\mathcal{W} \models \phi} P(\mathcal{W}) \qquad \text{where} \qquad P(\mathcal{W}) = \prod_{t \in \mathcal{W}} p(t) \cdot \prod_{t \in \mathcal{T} \setminus \mathcal{W}} 1 - p(t) \tag{7}$$

In practice, we can compute the probability $P(\phi)$ – in many cases – directly on the structure of the lineage formula ϕ. Let $Tup(\phi) \subseteq \mathcal{T}$ denote the set of tuples occurring in ϕ. Then, the following efficient computations can be employed [SOCK11]:

Definition			Condition
$P(I)$	$:=$	$p(I)$	$I \in \mathcal{T}$
$P(\bigwedge_i \phi_i)$	$:=$	$\prod_i P(\phi_i)$	$i \neq j \Rightarrow Tup(\phi_i) \cap Tup(\phi_j) = \emptyset$
$P(\bigvee_i \phi_i)$	$:=$	$1 - \prod_i (1 - P(\phi_i))$	$i \neq j \Rightarrow Tup(\phi_i) \cap Tup(\phi_j) = \emptyset$
$P(\neg\phi)$	$:=$	$1 - P(\phi)$	

(8)

Example 9. *Considering the lineage formula of DeNiro and Abbot's wedding in Figure 2(a), we obtain* $P((I_6 \wedge I_2) \vee (I_1 \wedge I_5)) = 1 - (1 - P(I_6 \wedge I_2)) \cdot (1 - P(I_1 \wedge I_5))$ *by applying the third line of Equation* (8)*. Then, the second and first line yield* $1 - (1 - p(I_6) \cdot p(I_2)) \cdot (1 - p(I_1) \cdot p(I_5)) = 0.92$*.*

If none of the cases of Equation (8) applies, we employ the following equation, called *Shannon expansion* [SOCK11], which is applicable to any propositional lineage formula:

$$P(\phi) := p(I) \cdot P(\phi_{[I/true]}) + (1 - p(I)) \cdot P(\phi_{[I/false]}) \tag{9}$$

Here, the notation $\phi_{[I/true]}$ for a tuple $I \in Tup(\phi)$ denotes that we replace all occurrences of I in ϕ by *true*. Repeated applications of Shannon Expansions, however, result in an exponential runtime.

Further Properties. As layed out in [DMT13, Dyl14] the sketched data model is closed and complete, i.e., it can express all instances of temporal probabilistic data. Furthermore, the data model can be extended by consistency constraints.

3 Top-k Querying

Instead of computing all answers to a query in a bottom-up manner as in Section 2, the focus of this section is computing only the top-k answers with the highest probabilities among all answers. For that, we start at the query and expand deduction rules until we reach the database tuples, which is called top-down grounding [AHV95].

First-Order Lineage. During top-down grounding, not all variables variables will be bound to constants. Hence, we now extend propositional lineage of Section 2 to first-order lineage [DTM13, Dyl14], which can contain variables and quantifiers. In contrast to propositional lineage, first-order lineage does not represent single query answers, but rather entire sets of answers. Each answer in such a set will be characterized by constants binding the query variables. To facilitate the construction of first-order lineage, we write all quantifiers of variables only occurring in the body of a deduction rule explicitly.

Example 10. *We equivalently rewrite the deduction rules of Equations* (10) *and* (11) *as follows:*

$$Wedding(P_1, P_2, T_b, T_e) \leftarrow \exists Did, D \begin{pmatrix} WeddingExtraction(P_1, P_2, Did, T_b, T_e) \\ \wedge\ FromDomain(Did, D) \end{pmatrix} \tag{10}$$

$$Divorce(P_1, P_2, T_b, T_e) \leftarrow \exists Did, D \begin{pmatrix} DivorceExtraction(P_1, P_2, Did, T_b, T_e) \\ \wedge\ FromDomain(Did, D) \end{pmatrix} \tag{11}$$

Note that in the above deduction rules all variables not occurring in the head literal, i.e., Did *and* D*, are bound by existential quantifiers.*

Grounding. Instead of starting at the database and instantiating the deduction rules to deduce new tuples, as before, we now begin at the query and in each step resolve a literal.

If the literal matches the head literal of a deduction rule, we replace it by the body of the rule. Otherwise, if the literal matches a database relation, we replace it by database tuples. Finally, if the no tuple and head literal matches, we substitute it by false. Whenever we bind a variable to constants $a_1, \ldots, a_n$, we employ the equivalence

$$\exists X \Phi \equiv \Phi_{[X/a_1]} \vee \cdots \vee \Phi_{[X/a_n]} \tag{12}$$

where $[X/a_i]$ substitutes the variable X by the constant a_i.

Example 11. *We consider the query* $Wedding(\textit{DeNiro}, P, T_1, T_2)$*, which asks for weddings of* DeNiro*. First, we replace the literal by the deduction rule body of Equation* (10)*:*

$$\exists Did, D\, WeddingExtraction(\textit{DeNiro}, P, Did, T_1, T_2) \wedge FromDomain(Did, D)$$

The above first-order lineage formula represents a set of query answers, one for each tuple of constants binding P*,* T_1*, and* T_2*. If we next exchange* $FromDomain(Did, D)$ *for the tuples of* I_5 *and* I_6 *of Figure 1, we obtain*

$$\begin{array}{c}(\, WeddingExtraction(\textit{DeNiro}, P, 1, T_1, T_2) \wedge I_5) \\ \vee (\, WeddingExtraction(\textit{DeNiro}, P, 2, T_1, T_2) \wedge I_6)\end{array} \tag{13}$$

where the disjunction results from Equation (12)*. The lineage formula still represents the same set of answers.*

Probability Bounds. Next, given a first-order lineage formula Φ, we construct two propositional formulas ϕ_{low} and ϕ_{up} whose probabilities then serve as lower and upper bound on Φ, respectively.

Definition 3. *Let* Φ *be a first-order lineage formula in negation normal form.*

1. *We construct the propositional lineage formula* ϕ_{up} *by substituting every literal* $R(\bar{X})$ *in* Φ *with* true *if* $R(\bar{X})$ *occurs positive in* Φ*, and by* false *otherwise.*
2. *We construct the propositional lineage formula* ϕ_{low} *by substituting every literal* $R(\bar{X})$ *in* Φ *with* false *if* $R(\bar{X})$ *occurs positive in* Φ*, and with* true *otherwise.*

Example 12. *We obtain* ϕ_{up} *from Equation* (13) *by setting the* WeddingExtraction *literals to* true*, which yields* $(true \wedge I_5) \vee (true \wedge I_6)$*. Hence,* $P((true \wedge I_5) \vee (true \wedge I_6)) = 1.0$ *is an upper bound.*

As shown in [DTM13, Dyl14], if $\phi_1, \ldots, \phi_n$ represent all query answers we would obtain by fully grounding Φ, then it holds that:

$$\forall i \in \{1, \ldots, n\} : \; P(\phi_{low}) \leq P(\phi_i) \leq P(\phi_{up})$$

Furthermore, with each step of the top-down grounding these bounds bounds converge monotonically to the probabilities of the answers. The resulting lower and upper bounds for all answer candidates can be plugged into any top-k algorithm [IBS08] that will then iteratively refine these lower and upper bounds until a termination condition is reached.

Pruning Answer Candidates. Given a set of answer candidates A_{top}, we can prune the answer candidate Φ, if $\forall \Psi \in A_{top} : P(\phi_{up}) \leq P(\psi_{low})$. Due to the monotonicity the lower bounds of all answer candidates in A_{top} can only increase, whereas the upper bound $P(\phi_{up})$ can only decrease. Hence, Φ never reaches a higher probability than any answer in A_{top}.

4 Learning Tuple Probabilities

Most works in the context of PDBs assume the database tuples along with their probabilities to be given as input. Also the preceding sections of this paper followed this route. Nevertheless, when creating, updating or cleaning a TPBD, the tuple probabilities have to be altered or even be newly created—in other words: they have to be *learned*, which is the topic of this section.

Example 13. *Assume we are given the TPDB of Figure 1 along with the deduction rules of Equations* (10) *and* (11)*. Figure 3 shows the resulting lineage formulas, where, however,*

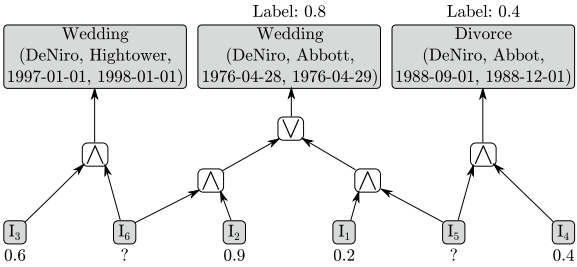


Figure 3: Lineage Formulas

the probabilities of the tuples of the FromDomain *relation are missing and marked with question marks. Instead, we are given labels for the deduced tuples from which we intend to learn the missing probabilities. For instance, a human assessor may have labeled the wedding of* DeNiro *and* Abbott *on April 28th 1976 to be 80% correct.*

Learning Problem. Formally, for a temporal probabilistic database $(\mathcal{T}, p, \mathcal{U}^T)$, we consider $\mathcal{T}_l \subseteq \mathcal{T}$ to be the set of database tuples for which we learn their probability values. That is, initially $p(I)$ is unknown for all $I \in \mathcal{T}_l$. Conversely, $p(I)$ is known and fixed for all $I \in \mathcal{T} \backslash \mathcal{T}_l$. To be able to complete $p(I)$, we are given labels in the form of pairs (ϕ_i, l_i), each containing a propositional lineage formula ϕ_i (i.e., a query answer) and its desired probability l_i. Now, we define the resulting learning problem.

Definition 4. *We are given a temporal probabilistic database $(\mathcal{T}, p, \mathcal{U}^T)$, a set of tuples $\mathcal{T}_l \subseteq \mathcal{T}$ with unknown probability values $p(I_l), I_l \in \mathcal{T}_l$, and a multi-set of given labels $\mathcal{L} = \langle(\phi_1, l_1), \ldots, (\phi_n, l_n)\rangle$, where each ϕ_i is a propositional lineage formula over $\mathcal{T}$ and each $l_i \in [0, 1] \subset \mathbb{R}$ is a probability for ϕ_i. Then, the* learning problem *is defined as:*

$$\begin{aligned} &\textit{Determine: } p(I_l) \in [0,1] \subset \mathbb{R} \textit{ for all } I_l \in \mathcal{T}_l \\ &\textit{such that: } \quad P(\phi_i) = l_i \textit{ for all } (\phi_i, l_i) \in \mathcal{L} \end{aligned}$$

Intuitively, we aim to set the probability values of the database tuples $I_l \in \mathcal{T}_l$ such that the labeled lineage formulas ϕ_i again yield the probability l_i. We want to remark that all probability values of tuples in $\mathcal{T} \backslash \mathcal{T}_l$ remain unaltered. Also, we note that the Boolean labels *true* and *false* can be represented as $l_i = 0.0$ and $l_i = 1.0$, respectively.

Example 14. *Formalizing the problem setting of Figure 3, we obtain* $\mathcal{T} = \{I_1, \ldots, I_6\}$, $\mathcal{T}_l = \{I_5, I_6\}$, $\mathcal{L} = \langle((I_6 \wedge I_2) \vee (I_1 \wedge I_5), 0.8), (I_5 \wedge I_4, 0.4)\rangle$.

Complexity. We next discuss the complexity of solving the learning problem. Unfortunately, it exhibits hard instances. First, computing $P(\phi_i)$ might require many Shannon expansions (see Equation (9)) causing exponential runtimes. But even for cases when all $P(\phi_i)$ can be computed in polynomial time (i.e., when Equation (8) is applicable), there are combinatorially hard cases of the above learning problem.

Lemma 1. *For a given instance of the learning problem of Definition 4, where all* $P(\phi_i)$ *with* $(\phi_i, l_i) \in \mathcal{L}$ *can be computed in polynomial time, deciding whether there exists a solution to the learning problem is* $\mathcal{NP}$*-hard.*

The above statement can be proven by encoding 3SAT into the learning problem [Dyl14].

Inconsistent Instances. After discussing the complexity of the learning problem, we characterize its solutions. First, there might also be *inconsistent* instances of the learning problem. That is, it may be impossible to define $p : \mathcal{T}_l \rightarrow (0, 1]$ such that all labels are satisfied.

Example 15. *If we consider* $\mathcal{T}_l := \{I_1, I_2\}$ *with the labels* $\mathcal{L} := \langle(I_1, 0.2), (I_2, 0.3), (I_1 \wedge I_2, 0.9)\rangle$*, then it is impossible to fulfill all three labels at the same time.*

Number of Solutions. From a practical point of view, there remain a number of questions regarding Definition 4. First, how many labels do we need in comparison to the number of tuples for which we are learning the probability values (i.e., $|\mathcal{L}|$ vs. $|\mathcal{T}_l|$)? And second, is there a difference in labeling lineage formulas that involve many tuples or very few tuples (i.e., $|\mathit{Tup}(\phi_i)|$)? Before we proceed to the theorem, recall that we can express probability computations $P(\phi)$ of a lineage formula ϕ as a polynomial following Equation (7). These polynomials have degree at most $|\mathit{Tup}(\phi)|$ as shown in [Dyl14].

Theorem 1. *If the labeling is consistent, the problem instances of Definition 4 can be classified as follows:*

1. *If* $|\mathcal{L}| < |\mathcal{T}_l|$*, the problem has infinitely many solutions.*
2. *If* $|\mathcal{L}| = |\mathcal{T}_l|$ *and the polynomials* $P(\phi_i) - l_i$ *have common zeros, then the problem has infinitely many solutions.*
3. *If* $|\mathcal{L}| = |\mathcal{T}_l|$ *and the polynomials* $P(\phi_i) - l_i$ *have no common zeros, then the problem has at most* $\prod_i |\mathit{Tup}(\phi_i) \cap \mathcal{T}_l|$ *solutions.*
4. *If* $|\mathcal{L}| > |\mathcal{T}_l|$*, then the polynomials* $P(\phi_i) - l_i$ *have common zeros, thus reducing this to one of the previous cases.*

The proof [Dyl14] of the above statement is accomplished via Bezout's theorem [DE10]. In general, a learning problem instance has many solutions, where Definition 4 does not specify a precedence, but all of them are equivalent. The number of solutions shrinks by adding labels to $\mathcal{L}$, or by labeling lineage formulas ϕ_i that involve fewer tuples in $\mathcal{T}_l$ (thus resulting in a smaller intersection $|\mathit{Tup}(\phi_i) \cap \mathcal{T}_l|$).

Example 16. *Employing Theorem 1 on the learning problem of Example 14, the third case applies, since* $|\mathcal{L}| = |\mathcal{T}_l|$ *and there are no common zeros. Hence, the theorem yields an upper bound of* $\prod_i |Tup(\phi_i) \cap \mathcal{T}_l| = |Tup((I_6 \wedge I_2) \vee (I_1 \wedge I_5)) \cap \mathcal{T}_l| \cdot |Tup(I_5 \wedge I_4) \cap \mathcal{T}_l| = |\{I_6, I_5\}| \cdot |\{I_5\}| = 2$. *The only solution, however, is* $p(I_5) = 0.4$ *and* $p(I_6) = 0.83$.

Solving the Learning Problem. Algorithmic solutions to the learning problem are beyond the scope of this paper, but we refer the interested reader to [Dyl14] for several gradient based approaches and a detailed experimental evaluation.

5 Conclusions

We presented a temporal probabilistic data model, devised how to compute the top-k query answers with the highest probabilities, and showed how to learn tuple probabilities in order to create or update probabilistic databases. For a more in-depth presentation of these aspects we refer the interested reader to [Dyl14].

References

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases*. Addison-Wesley, 1st edition, 1995.

[BDSH+08] Omar Benjelloun, Anish Das Sarma, Alon Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB Journal*, 17(2):243–264, March 2008.

[DBG12] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. SIGMOD, pages 433–444. ACM, 2012.

[DE10] Alicia Dickenstein and Ioannis Z. Emiris. *Solving Polynomial Equations: Foundations, Algorithms, and Applications*. Springer, 1st edition, 2010.

[DMT13] Maximilian Dylla, Iris Miliaraki, and Martin Theobald. A Temporal-Probabilistic Database Model for Information Extraction. *VLDB*, 6(14):1810–1821, 2013.

[DTM13] Maximilian Dylla, Martin Theobald, and Iris Miliaraki. Top-k query processing in probabilistic databases with non-materialized views. ICDE, pages 122–133, 2013.

[Dyl14] Maximilian Dylla. *Efficient Querying and Learning in Probabilistic and Temporal Databases*. PhD thesis, Saarland University, Saarbrücken, Germany, 2014.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys*, 40(4):11:1–11:58, October 2008.

[Jen00] Christian S. Jensen. *Temporal Database Management*. PhD thesis, Aalborg University, Aalborg, Denmark, April 2000.

[SOCK11] Dan Suciu, Dan Olteanu, R Christopher, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool, 1st edition, 2011.

Industrieprogramm

Events, Processes, Analytics

Pragmatisches Business Activity Monitoring für Workflow Management-Systeme in einer SOA

Karin Büscher, Andreas Hausotter, Arne Koschel, Jan Seidel

Fakultät Wirtschaft und Informatik
Hochschule Hannover
Ricklinger Stadtweg 120
30459 Hannover
karin.buescher@web.de
andreas.hausotter@hs-hannover.de
akoschel@acm.org
janseidel@t-online.de

Abstract: Um im Wettbewerb zu bestehen, müssen Unternehmen ihre durch Anwendungssysteme (semi-) automatisierten Geschäftsprozesse flexibel ändern können. Hier sind u.a. Workflow-Management-Systeme (WfMSe) in Kombination mit Service-orientierten Architekturen (SOAn) einsetzbar. Das WfMS dient der Steuerung von Service-Aufrufen innerhalb von Geschäftsprozessen. Mittels Geschäftsprozess-Controlling werden Prozesse analysiert und durch Geschäftsprozess-Monitoring (GPM) während der Prozessausführung anfallenden Informationen und auftretenden Ereignisse überwacht, dies auch zeitnah mittels Business Activity Monitoring (BAM). Dieser Artikel liefert hier Beiträge durch eine auf einem Data-Warehouse-Ansatz basierende Architektur für BAM, die einige Nachteile „klassischer" BAM-Werkzeuge vermeidet und auch GPM-Anforderungen erfüllt. Ein größeres Fallbeispiel aus dem Versicherungswesen zeigt die Tragfähigkeit der erarbeiteten Konzepte.

1 Einleitung

Prozess- und Service-Orientierung sind aktuelle Konzepte zur Gestaltung der IT-Landschaft in Unternehmen. Dieser Artikel betrachtet ausgewählte Fragestellungen, die sich bei der Umsetzung dieser Konzepte mittels Workflow-Management-Systemen und deren Kombination mit Service-orientierten Architekturen ergeben.

1.1 Workflow-Management-Systeme und Service-orientierte Architekturen

Workflow-Management-Systeme unterstützen Unternehmen bei der Koordination ihrer Geschäftsprozesse nebst zugehörigen Datenbeständen (vgl. [GHS95] und [Ho95]). Die Steuerung von Workflow-Client-Anwendungen (Client Applications) wird größtenteils durch ein WfMS vorgenommen. Individuen erhalten ihre Aufgaben (Tasks) aus einem

zentralen Arbeitskorb (Worklist), der durch die Geschäftsprozesse gefüllt wird (vgl. [Ho95]).

Vielversprechend ist die Kombination von WfMS mit Service-orientierten Architekturen. Eine Service-orientierte Architektur (SOA) ist das konzeptuelle Ergebnis der Aufteilung des Gesamtgeschäfts eines Unternehmens in Geschäftsdomänen (vgl. [Er05], [OA06], [KBS07] und [En08]). Entsprechend bietet sie den Zugriff auf Geschäftsfunktionen und -daten des Unternehmens, technisch bereitgestellt in Form von Services. Ein Service stellt somit eine klare Funktionalität hinter einer wohl definierten Schnittstelle bereit, dies in einer meistens logisch und technisch verteilten Systemumgebung.

Häufig wird die Implementierung einer SOA ausschließlich mit dem Einsatz von Web Services assoziiert (vgl. [Er05] und [[LN05]). Folglich werden dann für Prozesse meist nur die Web Services-Standards WSBPEL, BPEL4PEOPLE und WS HumanTask betrachtet (vgl. [OA07], [Kl05] und [Ag07]).

Allerdings greift diese Sichtweise in der Praxis oft zu kurz. Unternehmen können oder wollen aus betrieblichen Gründen keine (reine) Web Services-Lösung einsetzen, z.B. weil sich eine solche Lösung nicht nahtlos in die gewachsene IT-Landschaft integrieren lässt. Gleichwohl soll die Anwendungslandschaft Service-orientiert gestaltet werden, dies aber auf Basis anderer Technologien.

Eine gut geeignete und immer noch Standard-basierende Technologiekombination stellen WfMS gemäß der Spezifikation der Workflow Management Coalition (vgl. [Ho95]) kombiniert mit Java Enterprise (Java EE) Application Server-Lösungen oder anderen Integrationstechnologien wie CORBA (vgl. [HV09] und [Du08]) oder De-facto-Standards wie .NET (vgl. [Sc08] und [Du02]) dar.

Im Fokus dieses Artikels stehen also Service-orientierte Architekturen, die auf Workflow-Management-Systemen und anderen Integrationstechnologien basieren.

1.2 Anwendungshintergrund: Deutsches Versicherungswesen

Die Tragfähigkeit der Kombination WfMS und SOA als solide und aktuelle Basis für die zentrale Software-Architektur eines Unternehmens wird mittlerweile als gegeben angesehen. Sie kommt beispielsweise auch in deutschen Versicherungsunternehmen zum Einsatz.

Standardisierte Geschäftsprozesse aus dem Versicherungswesen stellen auch das Anwendungsumfeld des vorliegenden Beitrags dar. Sie sind in Deutschland mit der Versicherungsanwendungsarchitektur (VAA, vgl.[GD13]) gegeben. Somit sind die in diesem Artikel behandelten Fragestellungen potentiell für alle Arten deutscher Versicherungsunternehmen bedeutsam. Der VAA entstammt der hier als Beispiel vorgestellte Standardprozess „Kulanz bearbeiten". Da die in diesem Beitrag erzielten Ergebnisse i.W. anwendungsneutral sind, lassen sie sich potentiell auch auf andere Anwendungsfelder übertragen.

Wird die Kombination von WfMS und SOA technisch betrachtet, so heißt das, WfMS können zur Implementierung von Geschäftsprozessen eingesetzt werden. Diese Prozesse

nutzen wiederum die Services einer SOA. WfMS stellen also Geschäftsprozess-Management-Funktionen bereit und die Services der SOA liefern die wesentlichen Ein- und Ausgaben sowie Berechnungsfunktionen hierzu.

1.3 Geschäftsprozess-Controlling, Geschäftsprozess-Monitoring und Business Activity Monitoring

WfMSe persistieren alle während der Prozess-Ausführung anfallenden Informationen in der so genannten „Audit-Trail-Datenbank". Sie schaffen damit die technische Voraussetzung für das sog. Geschäftsprozess-Controlling (Business Process Controlling, BPC). Hierbei werden die gesammelten Informationen abgeschlossener Prozess- und Aktivitätsinstanzen aufbereitet, aggregiert und analysiert und betriebswirtschaftlich motivierte Kennzahlen (Key Performance Indicator, KPI) generiert.

Primäres Ziel des Geschäftsprozess-Controlling ist die kontinuierliche Optimierung der Prozessmodelle im Rahmen des Geschäftsprozess-Lebenszyklus. Dieser umfasst die Phasen Geschäftsprozess-Analyse und -Design, Workflow-Implementierung, Workflow-Ausführung und Workflow-basiertes Controlling (vgl. [Mu02]).

Über die im Rahmen des Geschäftsprozess-Controlling vorgenommene Analyse und Aufbereitung der Audit-Trail-Daten hinaus sind auch die aktuell in Ausführung befindlichen Prozess- und Aktivitätsinstanzen von besonderem Interesse. Fragestellungen wie z.B. „Wie lassen sich kritische Zustände, etwa die Schwellwertüberschreitung der Prozessdurchlaufzeit oder der Liegezeit einer Aktivitätsinstanz, rechtzeitig erkennen?" und „Welche Aktionen sind infolge kritischer Zustände auszuführen?" stehen per definitionem nicht im Fokus des Geschäftsprozess-Controlling. Sie lassen sich vielmehr dem sogenannten Geschäftsprozess-Monitoring (Business Process Monitoring, BPM) zuordnen, das die zeitnahe Überwachung aktiver Prozessinstanzen zum Gegenstand hat, um diese zielgerichtet zu beeinflussen und schnell auf auftretende Probleme und Engpässe reagieren zu können.

Damit ist das Geschäftsprozesse-Monitoring konzeptionell Teil des – allgemeineren – Business Activity Monitoring (BAM). Bei diesem von Gartner im Zusammenhang mit Enterprise Application Integration (EAI) geprägten Begriff handelt es sich um die Echtzeiterfassung kritischer Performance-Indikatoren mit dem Ziel, die Geschwindigkeit und Effektivität von Geschäftsfunktionen zu verbessern (vgl. [HF02]). Im Gegensatz zum Prozessmonitoring werden also bei BAM nicht nur Prozessinstanzen, sondern auch Middleware-Systeme – etwa Workflow-Management-Systeme und Datenbanksysteme – und Backend-Anwendungen einer Anwendungslandschaft berücksichtigt.

Geschäftsprozess-Controlling und -Monitoring unterstützen ein kontinuierliches Geschäftsprozessmanagement, das einerseits steuernde Eingriffe in die Prozessabläufe und andererseits deren agile und flexible Anpassung an geänderte Marktbedingungen ermöglicht (vgl. [Be07]). Zusammen mit dem Business Activity Monitoring liefert es damit einen wesentlichen Beitrag zur Sicherstellung der Wettbewerbsfähigkeit der Unternehmen.

BAM-Werkzeuge namhafter Hersteller sind auf dem Markt verfügbar, so z.B. von IBM, Oracle, Microsoft, TIBCO, SAP und der Software AG (vgl. [WSG07]). Da jedoch BAM

im Kontext von EAI entwickelt wurde, zeichnet es sich durch eine hohe Komplexität seiner Komponenten und eine nicht immer zufriedenstellend zu lösende Integration in vorhandene System- und Anwendungslandschaften aus. Als ggfs. problematisch ist der hohe Ressourcenbedarf anzusehen. Er resultiert vor allem aus dem Anspruch, kritische Anwendungs- und Systemzustände in Echtzeit zu erfassen und zu propagieren.

1.4 Ergebnisse

Da einerseits das Monitoring von Anwendungssystemen und Systemplattformen häufig durch spezialisierte Lösungen – etwa HP Open View – übernommen wird, andererseits eine Echtzeit-Überwachung aktiver Prozess- und Aktivitätsinstanzen im Rahmen des Geschäftsprozess-Monitoring oftmals nicht erforderlich ist, ist der BAM-Ansatz nach Gartner (vgl. [HF02]) für Unternehmen jedoch vielfach nicht die Methode der Wahl.

Hier setzen die Forschungsbeiträge der vorliegenden Arbeit an, die in Kooperation mit IT-Abteilungen aus mehreren mittelgroßen deutschen Versicherungsunternehmen entstanden sind. Die auf einem Data-Warehouse-Ansatz basierende Lösung erfüllt einerseits die Anforderungen an ein zeitnahes Geschäftsprozess-Monitoring; sie vermeidet andererseits aber den hohen Ressourcenbedarf „klassischer" BAM-Werkzeuge. Im Einzelnen sind die Beiträge dieses Artikels:

- Architekturkonzepte für ein Geschäftsprozess-Monitoring in Service-orientierten Architekturen mit Workflow. Alle Alternativen nutzen dabei die mit Informationen aus der Audit-Trail-Datenbank des WfMS befüllte „Foundation Database" (vgl. [Ha11]).
- eine Bewertung der Konzepte mit den Methoden der Nutzwertanalyse – unter Berücksichtigung der Wiederverwendbarkeit, Angemessenheit und des Realisierungsaufwands – und die anschließende Auswahl einer optimalen Alternative
- die prototypische Umsetzung des ausgewählten Architekturkonzepts
- die Evaluierung der realisierten Alternative anhand eines umfangreichen Fallbeispiels aus der VAA, dem Geschäftsprozess „Kulanz bearbeiten".

Für den von uns erstellten BAM-Prototypen wurden mehrere Leistungsmessungen durchgeführt, die u.a. ergaben, dass die Umsetzung in der jetzigen Form nicht für echtzeitnahe Anwendungsfälle geeignet ist. Komplette Durchläufe von der Ereigniserzeugung bis zur abschließenden Auswertung durch die Regelverarbeitung können auch im mehrstelligen Sekundenbereich liegen. Dies war jedoch aus Sicht der Kooperationspartner für deren Anwendungsbereiche in der Regel unkritisch und wurde für sie von der Flexibilität und Offenheit des Gesamtsystems mehr als aufgewogen.

Dennoch untersuchen wir in aktuell laufenden Arbeiten von uns auch den Transfer unserer Ergebnisse in Hybrid Cloud-basierte Umgebungen, wovon wir uns neben betrieblichem Optimierungspotential auch Performanz-Vorteile durch Parallelisierung versprechen.

Der Rest dieses Artikels ist wie folgt aufgebaut: Kapitel 2 behandelt verwandte Arbeiten. Die Kapitel 3 und 4 beschreiben das Anwendungsszenario und die Architekturkonzepte.

Kapitel 4 hat die prototypische Umsetzung der optimalen Alternative und die Evaluierung der Lösung zum Gegenstand. Zusammenfassung und Ausblick schließen diesen Artikel ab.

2 Verwandte Arbeiten

Neben der bereits einleitend durchgeführten Abgrenzung unserer Arbeiten zu Web Services-basierten Ansätzen wie WSBPEL etc. gibt es eine Reihe weiterer zum vorliegenden Artikel verwandte Veröffentlichungen, auf die im Folgenden näher eingegangen werden soll.

Allgemeine WfMS-Beiträge betrachten generelle WfMS-Konzepte und entsprechende Implementierungsaspekte (vgl. [GHS95], [De06] und [Ho95]). Sie liefern damit eine konzeptuelle Basis für weitergehende Forschungen. Demzufolge werden die hier vorgestellten Arbeiten in den konzeptuellen Kontext des Workflow-Referenzmodells der Workflow Management Coalition (WfMC, vgl. [Ho95]) eingeordnet. Diese Beiträge liefern jedoch nur allgemeine Konzepte; naturgemäß erreichen sie nicht den Spezialisierungsgrad der vorliegenden Arbeit.

In einem eng verwandten Kontext stehen Arbeiten, die Workflow-Management im SOA-Umfeld betrachten. So wird ein dezentralisiertes WfMS-Modell nebst Infrastruktur in [WML08] behandelt. Unsere BPM- und BAM-Architektur könnte eine nützliche Erweiterung derartiger Ansätze darstellen. Wie bereits erwähnt, sind die Konzepte unserer Arbeit übertragbar und damit vielleicht sogar geeignet, Web Services-Standards zu erweitern. Auf der anderen Seite könnten unsere Forschungen auch von Konzepten innerhalb der WSBPEL (vgl. [OA07]) und verwandten Spezifikationen profitieren.

Weitere verwandte Arbeiten betrachten die Kombination von WfMS und Data Warehouses. Bei [Mu01] beispielsweise wird ein WfMS und ein Data Warehouse für prozessorientierte Management-Informations-Systeme kombiniert und eine Taxonomie zur Auswertung der Audit-Trail-Datenbank sowie Komponenten zur Prozessüberwachung vorgestellt. Allerdings liegt bei diesem Artikel der Fokus – im Sinne eines BPC – eher auf der Analyse abgeschlossener Prozess- und Aktivitätsinstanzen und weniger auf der zeitnahen Reaktion auf kritische Prozesszustände – also BPM oder allgemeiner BAM – wie in unseren ForschungsarbeitenText.

Ebenfalls im Kontext stehende Arbeiten widmen sich betrieblichen und konzeptionellen Aspekten des Business Activity Monitoring. Bei [We09] wird eine Methodik zur Korrelation von KPIs und Quality of Service (QoS)- bzw. Prozessmetriken entwickelt. Die Arbeit von [Pe08] wiederum stellt einen semantischen Ansatz zur Ableitung betrieblich relevanter Informationen aus technischen Low-Level-Informationen der Audit-Trail-Datenbank vor.

Mehrere Veröffentlichungen legen den Fokus auf die Ereignisverarbeitung. So wird in [JFK04] ein regelbasiertes Framework für das BAM und in [JP06] die Ereignisverarbeitung in einer SOA für das BAM auf der Grundlage der SOA Lösung

SOPware der Deutschen Post vorgestellt. Ferner ist die Arbeit von [Gr06] zu erwähnen, da sie der fachlichen Domäne der Versicherungsunternehmen zuzurechnen ist. Die Autoren gehen auf die Neugestaltung der Prozesslandschaft nach SOA-Prinzipien ein und fokussieren dann auf die Verarbeitung von BPEL-Events und solchen, die aus einer "Event Cloud" durch ein "Complex Event Processing (CEP)"-System verarbeitet werden. Eine weitere Fallstudie diskutiert die Stärken und Schwächen verschiedener Performance-Management-Tools und stellt Architekturen für das Measurement und das Business Activity Monitoring vor (vgl. Ku05]).

Zusammengefasst könnten obige Beiträge in Teilgebieten der künftigen Ergänzung der vorliegenden Arbeit dienen, z.B. im Bereich der KPIs, QoS- bzw. Prozessmetriken. Umgekehrt könnten Arbeiten aus dem WfMS, BPC-, BPM- und BAM-Kontext ggf. auf unseren Arbeiten aufbauen.

Keine dieser Arbeiten stellt jedoch eine auf einem Data-Warehouse-Ansatz basierende integrierte Lösung für das Geschäftsprozess-Controlling, Geschäftsprozess-Monitoring und Business Activity Monitoring in Service-orientierten Architekturen mit WfMS bereit

3 Anwendungsszenario

3.1 Szenario

Die prototypische Implementierung des in diesem Artikel beschriebenen Business Activity Monitoring wurde einer umfassenden Evaluation unterzogen. Hierzu wurde ein Geschäftsprozess aus der Versicherungswirtschaft herangezogen und mit Hilfe einer Workflow-Client-Anwendung zur Bearbeitung von Aufträgen, die eine Benutzerinteraktion erfordern, umgesetzt.

3.2 Geschäftsprozess „Kulanz bearbeiten"

Ein Standardgeschäftsprozess der Versicherungswirtschaft ist „Kulanz bearbeiten", der innerhalb der VAA des Gesamtverbandes der Deutschen Versicherungswirtschaft (GDV) spezifiziert ist (vgl. [GD13]).

Im Verlauf dieses Prozesses werden zunächst die Vorschäden sowie die gezahlten Prämien des Versicherten ermittelt. Anschließend wird daraus der sogenannte Rentabilitätswert ermittelt. Bereits nach diesem Prozessschritt kann eine Ablehnung der Kulanzgewährung erfolgen. Sollte das nicht der Fall sein, wird ermittelt, ob das Vertragskonto des Versicherten ein negatives Saldo aufweist. Tritt dieser Fall ein, erfolgt ebenfalls eine Ablehnung. Andernfalls wird der zur Kulanz gemeldete Schaden mit einem festgelegten Wert verglichen. Sollte die Schadenshöhe unter diesem Wert liegen, wird die Kulanzhöhe errechnet und die Kulanz gewährt. Wenn die Schadenshöhe über diesem Wert liegt, erfolgt eine individuelle Beurteilung des Versicherten. Nach dieser Beurteilung wird

entschieden, in welcher Höhe eine Kulanz gewährt wird. Abschließend erfolgt in allen Fällen auch eine Korrespondenz mit dem Versicherten.

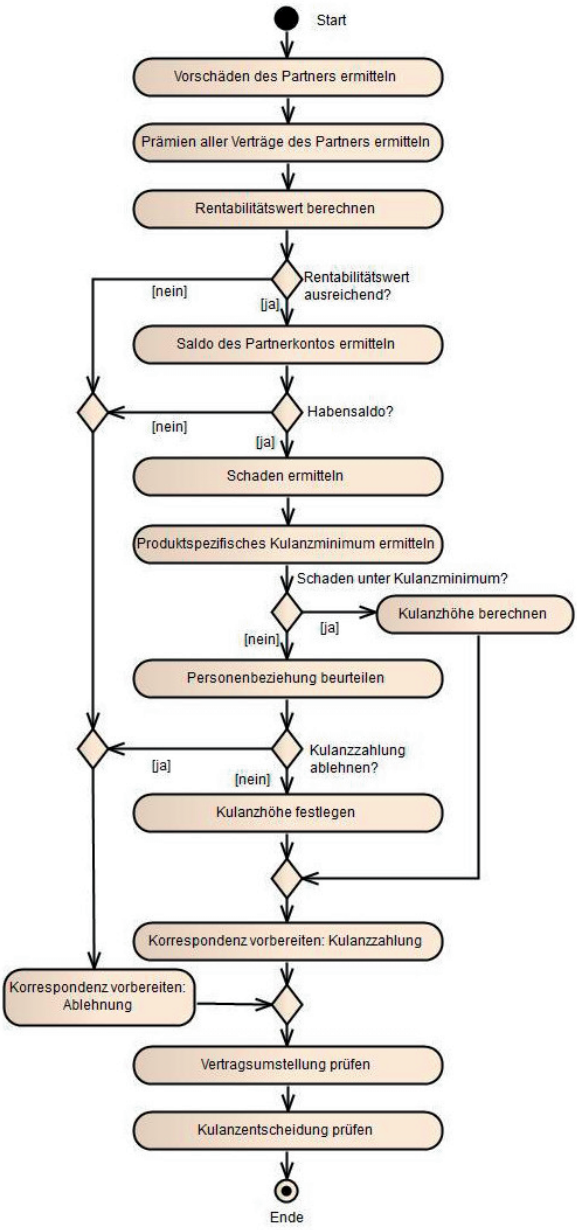


Abbildung 1: Geschäftsprozess „Kulanz bearbeiten“

Des Weiteren wird geprüft, ob eine Vertragsumstellung evtl. sinnvoll ist. Außerdem wird die Kulanzentscheidung abermals überprüft. Eine vereinfachte Darstellung des Prozesses wird im Beispiel „Kulanzprozess" (vgl. Abbildung 1) gezeigt.

3.3 Service-orientierte Realisierung

Das Anwendungsszenario, das diesem Beitrag zugrunde liegt, wurde aus Gründen der Wiederverwendbarkeit, Flexibilität und Agilität Service-orientiert gestaltet. Abbildung 2 zeigt den relevanten Ausschnitt aus der Service-orientierten Anwendungslandschaft.

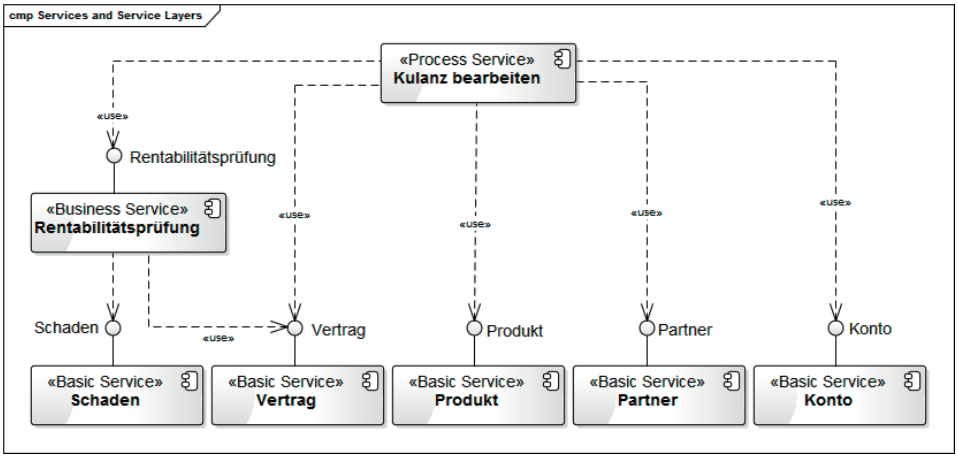


Abbildung 2: Services und Service Layers

„Schaden“, „Vertrag“, „Produkt“, „Partner“ und „Konto“ stellen datenzentrierte Basic Services dar, die die von den Backend-Systemen bereit gestellte Funktionalität über ihre Schnittstellen anbieten. „Rentabilitätsprüfung“ ist ein Business Service, der komplexe Rentabilitätsberechnungen durchführt und auf die Dienste der Services „Vertrag“ und „Schaden“ zugreift. Der Service „Kulanz bearbeiten“ übernimmt als Process Service die Orchestrierung.

Da aus betrieblichen Gründen eine reine Web Services-Lösung für die Kooperationspartner keine Option darstellte, kamen zur Realisierung der SOA-Alternative, Standardbasierende Technologiekombinationen wie Java Enterprise (Java EE) Application Server-Lösungen und ein Workflow-Management-System (WfMS) zur Anwendung.

So wurden „Schaden“, „Vertrag“, „Produkt“, „Partner“, „Konto“ und „Rentabilitätsprüfung“ als Java EE Stateless Session Beans realisiert. Da die Fachlogik, nach denen die Prüfung der Rentabilität erfolgt, häufigen Änderungen unterzogen ist, soll aus Flexibilitätsgründen in nachfolgenden Forschungsarbeiten der Service „Rentabilitätsprüfung“ mittels eines regelbasierten Ansatzes umgesetzt werden. Der Service „Kulanz bearbeiten“ wurde als Prozess modelliert. Die Instanziierung der im XPDL-Format vorliegen-

den Prozessbeschreibungen und deren Ausführung erfolgt durch die Workflow Engine des WfMS.

4 Business Activity Monitoring (BAM)

4.1 Grundlagen des BAM

Der Begriff Business Activity Monitoring wurde im Jahr 2003 durch die Gartner, Inc. geprägt (vgl. [HF02]). Business Activity Monitoring stellt dabei das Erfassen, Analysieren, Melden und Alarmieren wichtiger Ereignisse in Echtzeit dar. Im Gegensatz zum Business Process Monitoring werden beim BAM nicht ausschließlich Geschäftsprozesse überwacht, sondern zum Beispiel auch Anwendungssysteme und Middleware (vgl. Abbildung 3).

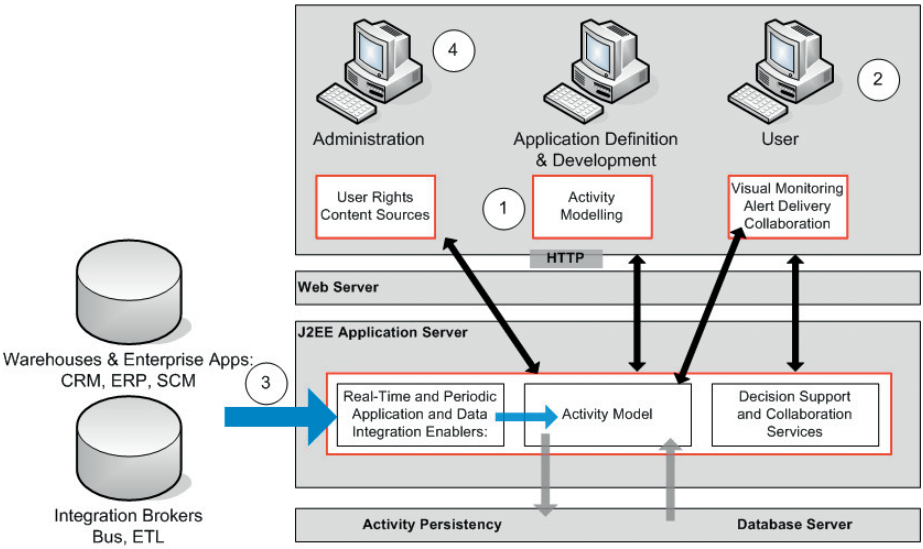


Abbildung 3: BAM nach Gartner

Im BAM muss zwar auf die anfallenden Daten in Echtzeit reagiert werden, dies führt aber nicht direkt zu einer Änderung am Ablauf des Geschäftsprozesses. Aus diesem Grund kann weiterhin ein Service getriebenes Architekturkonzept wie SOA eingesetzt werden. Allerdings müsste dieses Architekturkonzept dafür um eine Komponente zum Entdecken von Ereignissen in Echtzeit erweitert werden, wie es zum Beispiel eine Event Driven Architecture (EDA) bietet. Grundsätzlich ist dies auch eine denkbare (hier nicht gewählte) Option, z.B. basierend auf Complex Event Processing (CEP)-Ansätzen.

4.2 Architekturkonzepte

Für die Umsetzung der geforderten Funktionen hätte theoretisch auch eine am Markt befindliche „Standardlösung" eingesetzt werden können. Dies wurde jedoch gleich zu Beginn von den Kooperationspartnern aus unternehmensstrategischen und den in der Einleitung genannten fachlich-technischen Gründen – hohe Komplexität der Komponenten, hoher Aufwand bei einer Integration in die vorhandene System- und Anwendungslandschaft, hoher Ressourcenbedarf – verworfen.

Als Anforderung wurde genannt, dass verschiedene einfache Ereignisse, z.B. der Start eines Prozesses bzw. einer Aktivität und der Abschluss eines Prozesses bzw. einer Aktivität überwacht werden sollen. Aus diesen wiederum lassen sich verschiedene komplexe Ereignisse ableiten wie z. B. die Bearbeitungsdauer eines Prozesses bzw. einer Aktivität oder auch die Anzahl verschiedener Prozessinstanzen während eines bestimmten Zeitraums.

Im ersten untersuchten Grobkonzept werden zu Beginn des Programmablaufs durch einen Administrator die Regeln festgelegt, die zur Bildung von komplexen Ereignissen und somit zum Überwachen von Kennzahlen benötigt werden (vgl. Abbildung 4).

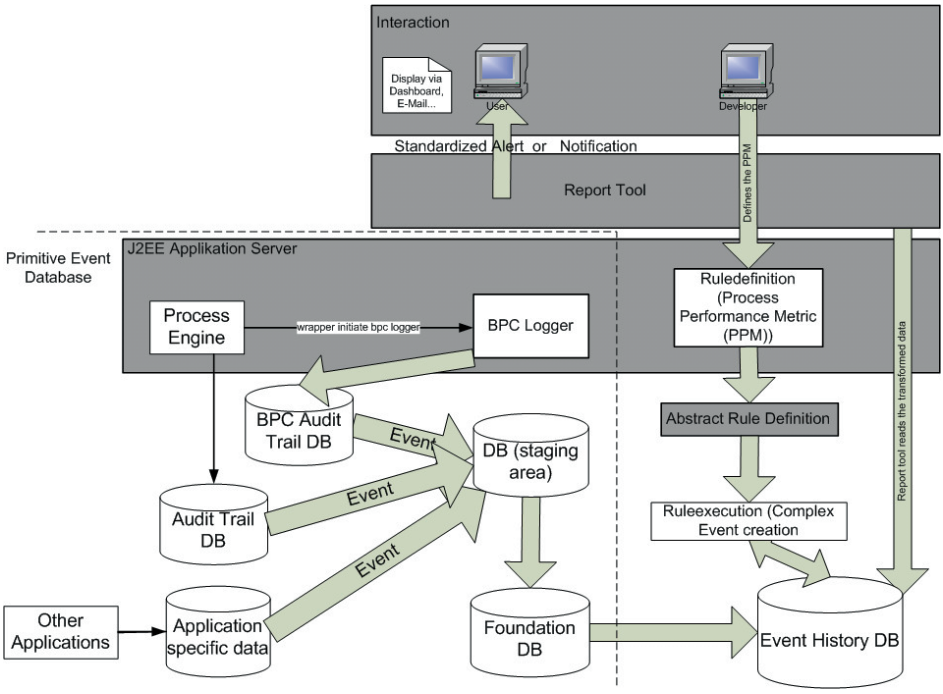


Abbildung 4: Grobkonzept mit BPC Logger und Foundation Database

Anschließend müssen die Regeln in eine abstrakte Regelsprache überführt werden. Diese abstrakte Regelsprache ermöglicht es, die Regelverarbeitung generisch zu halten. Die

mittels der abstrakten Regelsprache formulierten Regeln werden dann im Anschluss mit Hilfe eines Adapters in die produktspezifische Regelsprache überführt. Dies könnte z.B. durch Datenbanktrigger realisiert werden. Dazu würde ein Adapter an der Komponente der abstrakten Regelerzeugung für jede definierte Regel eine Implementation eines Triggers erzeugen.

Durch die Prozess Engine findet dann die Erzeugung von Ereignissen statt: So stößt die Engine über Wrapper den BPC Logger an. Dieser reichert die Daten um weitere Informationen an und speichert diese in der BPC Audit Trail DB. Aus dieser DB, der Audit Trail DB und eventuell aus applikationsspezifischen Datenbanken werden die Daten mittels ETL-Prozess in die Staging Area DB überführt und von dort durch einen ETL-Prozess in die Foundation DB geladen. Um dann einheitlich strukturierte Ereignisse in der Event History DB speichern zu können, werden die Daten erneut mittels ETL-Logik aus der Foundation DB extrahiert, transformiert und in die Event History DB geladen.

Durch die Komponente Ruleexecution werden die in der Event History DB gespeicherten Ereignisse gemäß der definierten Regeln auf Ereignismuster durchsucht und eventuell komplexe Ereignisse erzeugt, die ebenfalls in der Event History DB gespeichert werden. Das Report Tool sucht nun alle zu meldenden Ereignisse aus der Event History DB und meldet sie an die dafür zuständigen Personen. Für die Meldungen kann zum Beispiel ein Dashboard verwendet werden (vgl. Abbildung 4).

Das zweite untersuchte Grobkonzept ist ähnlich dem oben beschriebenen Grobkonzept modelliert. Der wesentliche Unterschied besteht darin, dass statt eines ETL-Prozesses zwischen der Foundation DB und der Event History DB ein Adapter zum Einsatz kommt.

Im dritten untersuchten Grobkonzept sollen die anfallenden Daten direkt in der Audit Trail DB und weiteren applikationsspezifischen Datenbanken gespeichert werden. Diese Datenbanken werden wiederum von Triggern überwacht. Von den Triggern werden die Ereignisse standardisiert und an die Regelverarbeitung übermittelt. Die Regelverarbeitung wendet die zuvor definierten Regeln an, um die gemeldeten Ereignisse zu verarbeiten und eventuell komplexe Ereignisse zu erzeugen. Von der Regelverarbeitung werden dann alle Ereignisse in der Event History DB gespeichert.

4.3 Bewertung der drei Grobkonzepte

Um eine Entscheidung zur Verwendung eines Grobkonzeptes treffen zu können, wurden diese mit Hilfe eines gewichteten Kriterienkatalogs bewertet. Die Grobkonzepte wurden dabei in sechs verschiedenen Kategorien bewertet. Diese Kategorien sind die Funktionalität, die Zuverlässigkeit, die Benutzbarkeit, die Effizienz, die Wartbarkeit und die Übertragbarkeit. Diese Kategorien wurden jeweils wiederum in mehrere Kriterien unterteilt.

Das erste Grobkonzept mit BPC Logger und Foundation Database hat eine im Vergleich zu den anderen Optionen bessere Bewertung erhalten. Dieser Entscheidung zugrunde liegt der höhere Reifegrad wegen der Wiederverwendung bestehender Komponenten und der geringere Realisierungsaufwand.

5 Prototypische Umsetzung und Evaluierung

Die Bewertung der drei Grobkonzepte auf der Grundlage des mit den Kooperationspartnern abgestimmten Kriterienkatalogs ergab eine deutliche Präferenz für das Konzept mit BPC Logger und Foundation Database (vgl. Abbildung 4), sodass in nachfolgenden Studien dieses konkretisiert und zu einem Feinkonzept weiterentwickelt wurde.

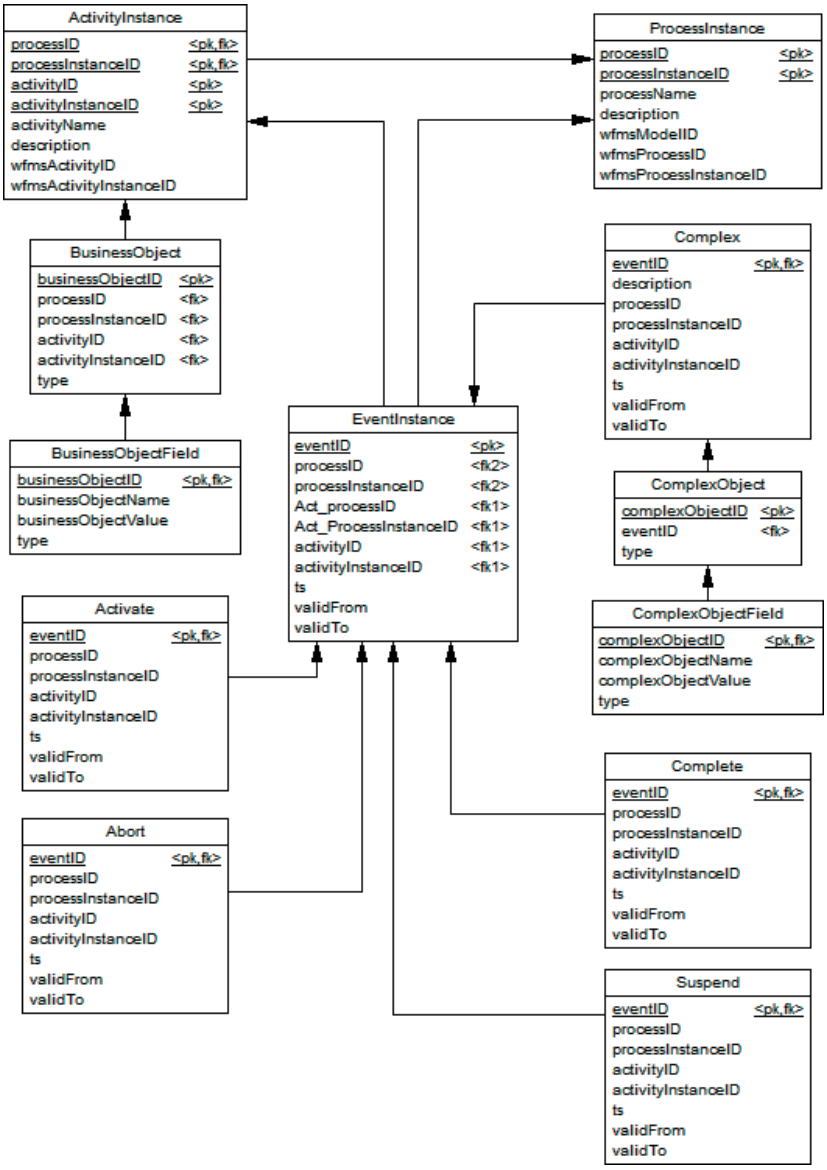


Abbildung 5: Datenbankschema der Foundation Database

In diesem Kapitel sollen zunächst die wesentlichen Konzepte zur Regelmodellierung und -verarbeitung vorgestellt und ein Einblick in die Implementierung gegeben werden. Abschließend wird auf Probleme, die sich während der Implementierung ergeben haben, eingegangen und eine Einschätzung der Laufzeiteigenschaften unserer Lösung vorgenommen.

5.1 Verarbeitung elementarer und komplexer Ereignisse

In der Event History DB werden elementare Ereignisse abgespeichert. Mittels eines Primärschlüssels kann eine Ereignisinstanz eindeutig identifiziert werden. Eine Spezialisierung der Ereignisinstanz wird über die Tabellen Activate, Complete, Abort, Suspend sowie Complex vorgenommen (vgl Abbildung 5).

Die Tabelle mit den komplexen Ereignissen wird erst im späteren Verlauf durch die auf der Event History DB arbeitenden Trigger befüllt. Um die Information, aus welchen elementaren Ereignissen ein komplexes Ereignis entstanden ist, nicht zu verlieren, wird diese in Einträgen der entsprechenden Tabellen ComplexObject in Verbindung mit der Tabelle ComplexObjectField hinterlegt. Dort kann jedoch nicht die Modellierungsvorschrift sondern nur die Zusammensetzung eines speziellen komplexen Ereignisses nachvollzogen werden. Die Modellierungsvorschrift muss in jenen Datenbanktriggern implementiert werden, die die komplexen Ereignisse erzeugen. Vorteilhaft ist bei dieser Umsetzung, dass bei neuen komplexen Ereignissen keine Anpassungen am Datenbankmodell erforderlich werden, sondern lediglich ein neuer Trigger implementiert werden muss.

Außerdem wird durch die Modellierung der Tabellen Complex, ComplexObject und ComplexObjectField vermieden, dass Daten von einem Trigger gelesen werden, um sie in einer Transaktion zu verarbeiten, während diese Daten noch von einer anderen Transaktion geändert werden und noch nicht bestätigt sind.

Durch diese Modellierung wird somit sichergestellt, dass jede Transaktion nach dem ACID-Prinzip korrekt abgearbeitet wird, da sich keine gleichzeitig stattfindenden Transaktionen gegenseitig beeinflussen.

Während der Entwicklung traten kurzzeitig Schwierigkeiten rund um die Datenkonsistenz auf. Durch lesende Datenzugriffe eines Triggers, während diese von einem anderen Trigger in einer Transaktion versucht wurden zu verändern, kam es zu Seiteneffekten. Durch die Anpassung im zugrundeliegenden Datenmodell konnte eine Vermeidung von Lese-Schreib-Konflikten sichergestellt werden.

5.2 Aufbau der eingesetzten Trigger zur Regelüberwachung

Da nach der Befüllung der Event History DB alle elementaren Ereignisse vorhanden sind, können im Anschluss auch die komplexen Ereignisse erzeugt werden. Mit jedem auftretenden elementaren Ereignis kann auch wieder ein komplexes Ereignis entstehen. Um diese Ereignismuster zu überwachen – also bei jedem Auftreten eines elementaren

Ereignisses eventuell auch ein komplexes Ereignis zu erzeugen – werden die Ereignismuster mittels Datenbanktriggern realisiert.

Die Durchführbarkeit dieses Vorhabens soll anhand eines beispielhaft mittels SQL implementierten Triggers aufgezeigt werden. Dafür wurde das komplexe Ereignis loggingNumberPassesOfActivityPerDay herangezogen. Dies beschreibt die Protokollierung der Anzahl der Durchläufe einer Aktivität pro Tag. Abbildung 6 zeigt den zeitlichen Ablauf des Triggers.

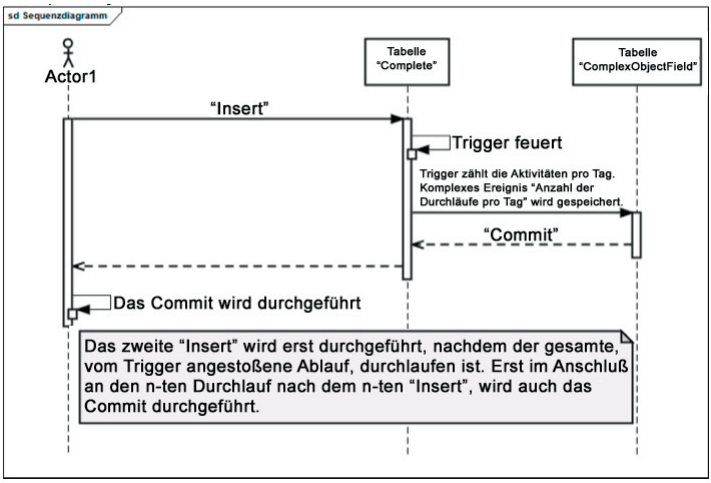


Abbildung 6Ablauf eines Triggers

Neben den komplexen Ereignissen, die aufgrund ihrer fachlichen Notwendigkeit definiert wurden, sollen auch auftretende Fehler durch ein komplexes Ereignis erkannt und verarbeitet werden. Dazu wurde ein Trigger definiert, der auf jede mögliche fehlerhafte logische Kombination von elementaren Ereignissen, wie z. B. ein elementares Ereignis, das vor einem anderen auftreten muss aber einen späteren Zeitstempel trägt, reagiert. Durch diesen Trigger werden somit fehlerhafte Eingaben durch Nutzer sowie fehlerhaft ausgeführte automatisch ablaufende Prozesse erkannt.

Um den erkannten Fehler zu protokollieren, erzeugt der Trigger wiederum ein komplexes Ereignis. In diesem Ereignis werden die fehlerhaften Parameter protokolliert.

5.3 Erzeugung der Trigger durch die Abstract Rule Definition

Damit spätere Anwender des Systems keine tiefer gehenden Datenbankkenntnisse besitzen müssen, um für jedes neu zu berücksichtigende Ereignismuster einen eigenen Trigger realisieren zu können, soll die Erzeugung der Trigger durch die Komponente - Abstract Rule Definition - erfolgen. Außerdem wird dadurch eine Entkopplung der zugrundeliegenden Technologie erreicht. Zur Nutzung dieser Komponente ist ausschließlich die Kenntnis einer abstrakten Regelsprache notwendig.

Die Erzeugung der Trigger findet über einen vordefinierten Ablauf statt. Dieser ist in der Komponente Abstract Rule Definition hinterlegt. Eine Verfeinerung dieser Komponente und ihr Zusammenspiel mit der Komponente Ruleexecution ist in Abbildung 7 zu sehen.

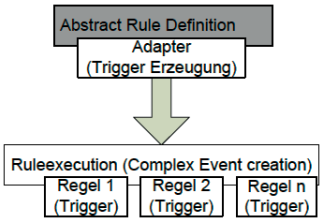


Abbildung 7: Prozess der Trigger-Erzeugung

Zu Beginn der Verarbeitung existiert ein XML-Dokument, deren Inhalt die abstrakte Definition der zu verarbeitenden Regel darstellt. Die Erzeugung dieses XML-Dokumentes erfolgt im Vorfeld manuell. In einer späteren Projektphase soll die Definition abstrakter Regeln durch ein graphisches Werkzeug erfolgen können.

Die Abstract Rule Definition stellt dabei eine Metasprache zur Beschreibung von Regeln dar. Diese Regeln sollen durch Regelcompiler in produktspezifische Regeln überführt werden. In dem entwickelten Feinkonzept wird die abstrakte Regeldefinition über den oben gezeigten Ablauf in Trigger überführt. Die Menge der Trigger bilden somit die Regeln in der Datenbank ab. Jede abstrakte Regel wird in einen Trigger in der Datenbank überführt, außerdem wird für jede Regel ein eigenes XML-Dokument angelegt. Die XML-Dokumente werden von einem Parser wie zum Beispiel JAXP geparst. Die daraus gewonnenen Informationen müssen von der Rule Builder-Komponente weiter verarbeitet und in Trigger-Funktionen überführt werden. Das Anlegen der Trigger in der Datenbank kann zum Beispiel über eine JDBC-Schnittstelle stattfinden.

5.4 Verwendung einer Abstrakten Regelsprache

In der Komponente Abstract Rule Definition muss eine abstrakte Regelsprache eingesetzt werden, um aus einem Ereignismuster einen Trigger zu erzeugen. Dieser Zwischenschritt dient dazu, die Ereignismuster unabhängig von der zugrundeliegenden Technologie entwickeln zu können. Ereignismuster werden definiert und in ein XML-Dokument, oder eine grafische Benutzeroberfläche eingegeben, ohne dass ein Wechsel des Datenbanksystems darauf Auswirkungen hat. Die abstrakte Regelsprache übernimmt also die Aufgabe vorgegebene Ereignismuster in einen datenbankspezifischen Trigger zu überführen (vgl. Abbildung 7). Eine mögliche Umsetzung einer abstrakten Regelsprache stellt RuleML (vgl. [BTW01]) dar.

Ebenso wie die Trigger muss auch die verwendete abstrakte Regelsprache einige Anforderungen erfüllen, um alle denkbaren Ereignismuster interpretieren und umwandeln zu können. Es ist nötig, dass mathematische Operatoren (+, -, *, /), logische Operatoren und gängige Vergleichsoperatoren direkt unterstützt oder durch die Verkettung mehrerer Bedingungen mittels eines UND-Operators gebildet werden können. Zudem müssen ein SUM()-Operator und ein CAUSES-Operator unterstützt werden.

5.5 Implementierung und Evaluierung

Die erarbeiteten Konzepte wurden durch eine prototypische Umsetzung evaluiert. Hierzu wurde eine Implementierung der Event History DB sowie der beschriebenen Trigger auf der Basis Der Oracle DB vorgenommen. ETL-Prozesse wurden mittels des Pentaho Data Integration Tools umgesetzt. Die Bestandteile wurden separaten Funktionstests unterzogen. Um die prototypische Umsetzung in Gänze zu evaluieren wurden umfangreiche Lasttests durchgeführt. Verwandt wurden hierfür virtuelle Maschinen, die wiederum auf mehreren herkömmlichen Arbeitsplatz PCs betrieben wurden. Die Leistungsmessung erfolgte auf Basis generierter SQL-Skripts für verschiedene fiktive Eingaben. Somit ließ sich sowohl ermitteln, welche Bearbeitungszeiten für einzelne Eingaben benötigt werden, als auch, wie viele Eingaben parallel erfolgen können, ohne das die Bearbeitungszeiten merkbar negativ beeinflusst wurden.

Die Dauer der Erzeugung einzelner Ereignisse wie auch die Bearbeitung vollständiger Workflows befand sich dabei im Millisekundenbereich. Insgesamt war die Bearbeitung von ca. 1.000 Workflows pro Minute möglich ohne die Bearbeitungszeiten negativ zu beeinflussen. Dabei belief sich die Datenmenge der abgearbeiteten Workflows inklusive der zugehörigen Ereignisse auf ca. 1 MB pro 1.000 Workflows.

Ein abschließender Performancevergleich mit am Markt befindlichen „Standardlösungen“ hat nicht stattgefunden, da der Einsatz einer solchen Lösung von den Kooperationspartnern nicht in Betracht gezogen wurde und dementsprechend keine Alternative darstellte. Auch wurden die zunächst entwickelten Grobkonzepte, nachdem sie verworfen worden sind, nicht umgesetzt, sodass auch hier keine vergleichenden Performancetests stattgefunden haben.

Zusammenfassend standen Performanceuntersuchungen nicht im Fokus unserer Arbeiten. Vielmehr besaßen für die Kooperationspartnern die Kriterien Wiederverwendung bestehender Komponenten, Flexibilität und Offenheit des Gesamtsystems eine deutlich höhere Priorität als etwa die Performance (vgl. Kapitel 4).

6 Zusammenfassung und Ausblick

Unternehmen stehen vor erheblichen Herausforderungen: Um im globalisierten Markt bestehen zu können, müssen in immer kürzer werdenden Zyklen vorhandene Geschäftsprozesse flexibel an sich schnell ändernde Anforderungen angepasst werden. Serviceorientierte Architekturen scheinen geeignet, diese Anforderungen an Flexibilität und Agilität erfüllen zu können. Technisch kann eine SOA zum Beispiel auf Basis von Web Services-Standards implementiert werden. Da SOA jedoch ein technologieneutrales Konzept ist, können gut auch andere technische Lösungen zum Einsatz kommen, die ggf. besser zur IT-Landschaft eines Unternehmens passen.

In unseren Arbeiten wurde deshalb eine SOA auf Basis des Java EE-Standards und einem Workflow-Management-System umgesetzt. Im SOA-Kontext realisieren dann die Prozessbeschreibungen die sogenannten Prozess-Services, die durch die Workflow En-

gine instanziiert und ausgeführt werden. Ein Geschäftsprozess-Monitoring unterstützt die Effizienzkontrolle der SOA. Im Themenkontext SOA, WfMS und Geschäftsprozess-Monitoring liegen folglich die Beiträge dieser Arbeit. Diese sind:

- verschiedene Architekturkonzepte für ein Geschäftsprozess-Monitoring in Service-orientierten Architekturen mit WfMS (vgl. Unterkapitel 4.2)
- eine Bewertung der Konzepte mit den Methoden der Nutzwertanalyse hinsichtlich Wiederverwendbarkeit, Angemessenheit und Realisierungsaufwand (vgl. Unterkapitel 4.3)
- ein Prototyp zur Umsetzung des gewählten Architekturkonzepts (vgl. Kapitel 5) sowie die Evaluierung der Ergebnisse anhand eines umfangreichen Fallbeispiels aus der VAA, dem Geschäftsprozess „Kulanz bearbeiten".

Auf Basis der erfolgreich erzielten Ergebnisse erweitern wir in aktuellen Arbeiten unsere Gesamtarchitektur um den Aspekt der Geschäftsregeln (Business Rules Management) und untersuchen die Verknüpfung der bisherigen Ergebnisse mit Cloud-basierten IT-Umgebungen.

Literaturverzeichnis

[Ag07] Agrawal, A. et al. (2007), Web Services Human Task (WS-HumanTask), Version 1.0, Technical report, Active Endpoints, Adobe, BEA, IBM, Oracle, SAP.

[Be07] Becker, M. (2007), Geschäftsprozess-Controlling auf der Basis von Business-Intelligence-Konzepten und Data-Warehouse-Systemen: Architekturen , Datenmodell, Vorgehensmodell und Anwendungsszenarien, Shaker: Aachen.

[BTW01] Boley, H., Tabet, S., Wagner, G. (2001), Design Rationale for RuleML: A Markup Language for Semantic Web Rules, DFKI GmbH, Kaiserslautern.

[De06] De Jong, P. (2006), Going with the Flow, in: ACM Queue, pp 26--32.

[Du08] Dunkel, J., Eberhart, A., Fischer, S., Kleiner, C., Koschel, A. (2008), Systemarchitekturen für Verteilte Anwendungen, Hanser, München

[En08] Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M. und Richter, J.-P. (2008), Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten, dpunkt.verlag, Heidelberg.

[Er05] Erl, T. (2005), Service-Oriented Architecture: Concepts, Technology, and Design, 5th edt., Prentice Hall, NJ.

[GD13] GDV (2001), VAA Final Edition. Das Objektorientierte Fachliche Referenzmodell Version 2.0, Berlin, http://www.gdv-online.de/vaa/vaafe_html/dokument/otrm.pdf, Abruf am 05.12.2013.

[GHS95] Georgakopoulos, D., Hornick, M., Sheth, A. (1995), An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, in: Distributed and Parallel DBS v. 3, p. 119--153, USA.

[Gr06] Greiner, T., Duster, W., Pouatcha, F., von Ammon, R. (2006), Business Activity Monitoring of Norisbank Taking the Example of the Application easyCredit and the Future Adoption of Complex Event Processing (CEP), in: Proc. IEEE Services Comp. Workshops SCW '06, pp 83.

[Ha11] Hausotter, A., Kleiner, C., Koschel, A., Zhang, D., Gehrken, H. (2011), Always Stay Flexible! WfMS-independent Business Process Controlling in SOA, in: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011, 15th IEEE International', pp 184–193.

[HF02] Hellinger, M., Fingerhut, S. (2002), Business Activity Monitoring: EAI Meets Data Warehousing, in: EAI Journal, pp 18--21.

[Ho95] Hollingsworth, D. (1995), The Workflow Reference Model, Wf Management Coalition, UK.

[HV09] Henning, M., Vinoski, S. (2009), Advanced CORBA Programming with C++, Add.Wesley, USA.

[JFK04] Jeng, J. J., Flaxer, D., Kapoor, S. (2004), RuleBAM: a rule-based framework for business activity management, in: Proc. IEEE Intl. Conference on Services Computing (SCC 2004), pp 262--270.

[JP06] Jobst, D., Preissler, G. (2006), Mapping clouds of SOA- and business-related events for an enterprise cockpit in a Java-based environment, in: PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java, ACM, NY, USA, pp 230--236.

[KBS07] Krafzig, D., Banke, K., and Slama, D. (2007), Enterprise SOA: Best Practices für Serviceorientierte Architekturen -- Einführung, Umsetzung, Praxis, mitp-Verlag, Heidelberg.

[Kl05] Kloppmann, M.et al. (2005), WS-BPEL Extension for People -- BPEL4People, Technical report, SAP AG, IBM Corp.

[Ku05] Kung, P., Hagen, C., Rodel, M., Seifert, S. (2005), Business process monitoring and measurement in a large bank: challenges and selected approaches, in: Proc. Sixteenth International Workshop on Database and Expert Systems Applications, pp 955--961.

[LN05] Lomow, G., Newcomer, E. (2005), Understanding Service-Oriented Architecture (SOA) with Web Services, Addison Wesley, USA.

[Mu01] zur Muehlen, M. (2001), Process-driven management information systems.

[Mu02] zur Muehlen, M. (2002), Workflow-based Process Controlling, Logos Verlag: Berlin.

[MWL08] Martin, D., Wutke, D., Leymann, F. (2008), Synchronizing control flow in a tuplespace-based, distributed workflow management system, in: Proceedings of the 10th international conference on Electronic commerce (ICEC '08). ACM, NY, USA.

[OA06] OASIS (2006), Reference Model for SOAs 1.0, 2006, http://docs.oasis-open.org/soa-rm/v1.0.

[OA07] OASIS (2007), WSBPEL -- Web Services Business Process Execution Language Version 2.0, OASIS Standard, 2007. -- combining data warehouses and workflow technology. in: Proc. 4th Intl. Conf. on EC Research (ICECR-4), B. Gavish, Ed., Dallas, TX, Nov 2001, pp 550--566.

[Pe08] Pedrinaci, C., Lambert, D., Wetzstein, B., van Lessen, T., Cekov, L., Dimitrov, M. (2008), SENTINEL: a semantic business process monitoring tool, in: OBI '08: Proc. 1st international workshop on Ontology-supported business intelligence, ACM, New York, NY, USA, pp 1--12.

[Sc08] Schwichtenberg, H.(2008), Microsoft .NET 3.5 - Crashkurs: Presentation, Communication, Workflow, Visual Basic 2008, Visual C\# 2008, ADO.NET 3.5, ORM, LINQ, WPF, WCF, Microsoft Press, München.

[We09] Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., Leymann, F. (2009), Monitoring and Analyzing Influential Factors of Business Process Performance, in: Proc. IEEE International Enterprise Distributed Object Computing Conference EDOC '09, pp 141--150.

[WML08] Wutke, D., Martin, D., Leymann, F. (2008), Model and infrastructure for decentralized workflow enactment, in: Proc. ACM SAC '08. ACM, NY, USA, pp 90--94.

[WSG07] Wagner, pp, Sonntag, M., Gostmann, H. (2007), Vergleich von Business Activity Monitoring Werkzeugen, Fachstudie, IAAS, Universität Stuttgart, Stuttgart.

Event-driven tracking of important sections of business processes

Christoph Heinz
Software AG
Darmstadt, Germany
christoph.heinz@softwareag.com

Abstract: Business processes are crucial in today's companies. They define for the different departments of a company the execution order of tasks as well as their interaction. Typically specific sections within a business process are essential for its successful completion and therefore deserve special attention. Such a section is typically associated with a completion constraint, e.g., a credit application has to be processed within 24 hours in order to meet SLAs agreed with the customer. The business user is highly interested in a real-time monitoring of those sections and constraints as it allows him to immediately tackle potential problems during process execution. We present a novel approach for tracking important sections of a business process using Complex Event Processing coupled with an Event-Driven Architecture. While the business process is executed, we continuously monitor its processing status to determine whether such a section is completed and its associated constraints have been fulfilled or violated. To allow for a more proactive reaction of the business user, we also present an early warning system which raises an alert if a not yet completed section runs the risk of violating a constraint.

1 Introduction

In modern companies business processes play a vital role as they clearly define how tasks in daily business are connected to each other and how they are processed. Business processes are used in virtually every industry and each department of a company. For example, there are processes that define the tasks for an approval of an insurance claim, an invitation of a job applicant for an interview, the shipment of an order to the customer, the authorization of a business travel. Each of those business processes consists of several tasks which are connected to each other, thereby modeling the basic sequence of steps from start to end of the process.

A business process can be defined in terms of a graphical model consisting of symbols and connections in between them, using for example the well-established BPMN modeling standard. The latter also defines how a model can be transferred into a process execution language. There are also other approaches available for defining an execution of business processes. Thus, business processes are available as models and transformed into executable instances. The execution of those process instances can be monitored in order to inform the business user about the current status of business processes he is interested in.

In the area of business process monitoring tools Software AG has recently introduced the notion of stages and associated milestones. A milestone is defined as one particular point in a process occurring at a specific point in time. For example, an order has been received at 15:25 or an email has been sent at 10:13. A stage is defined by a start and an end milestone and a condition how these two milestones relate to each other. Most prominent type of such stages is a temporal stage. The corresponding temporal condition demands that the timespan between start and end milestone has to be either greater or less than a predefined timespan. An exemplary stage could define that the timespan between receiving an order, which denotes the start milestone, and shipping the goods, which denotes the end milestone, has to be less than 12 hours. Thus, a stage allows the user to formally define the most important sections of a business process and corresponding completion criteria.

In this work we tackle a real-time tracking of those stages. Prerequisite is that the IT component executing the process instances emits status information in case of process instance changes, e.g., the order step has been started at 5:00. By evaluating this process status information in a continuous fashion, a corresponding process stage tracking component can directly inform the user that a stage has been started, breached, or completed. Using real-time processing capabilities, this stage information can be derived with minimum latency, e.g., the user is notified immediately if a stage has been breached. This enables the user to monitor crucial sections of his business processes.

We introduce the notion of process stage tracking with a focus on the proactive handling of stage violations. As described before the user is notified when a stage finally has breached or completed. A further extension is to provide early warnings when a not yet completed stage potentially might be violated.

The following enumeration sketches core problems a process stage tracking component has to deal with and our proposed solutions:

Detection of stages at risk: Recent process tracking capabilities allow to send notifications of stages being started, breached or completed in a real-time manner. Thus, the user knows immediately when a process stage has breached for instance. He can react to this situation and trigger consecutive actions. However, the user can no more take corrective actions to proactively avoid that a stage breaches in the near future. Therefore, the proposal is to provide an early warning to the user that a successful stage completion is at risk. While the process is executing and the current execution of a process instance is in between start and end milestone of a stage, the probability that the remaining steps until the end milestone take longer than the remaining time to stage completion is determined. If that probability is too high, the corresponding stage is classified as being at risk and an alert is sent.

Definition of potential stage violations: The detection of potential stage violations requires to estimate the remaining time from the current position in the process execution to the end milestone. For that purpose each step in the process has an associated statistical model which models the timespan this step typically takes. These models are continuously updated by incorporating recently measured step cycle times. By combining these step cycle models we estimate the remaining time to the end milestone. These estimates provide the user means to assess a potential violation of the stage.

Real-time detection of potential violations: In order to ensure a maximum transparency for the user potential stage violations are computed in a real-time manner. Provided that the process execution engine reports process instance changes directly, Complex Event Processing technology coupled with an Event-Driven Architecture [EN10], [SC09] can be leveraged to determine stage violations in real-time. Each time a process instance change occurs, the probability for violating the process stage is directly computed. Such a result can then be forwarded to the user. For example, a corresponding visual cockpit can highlight for a selected process model the corresponding stages being at risk. Additionally, an email can be sent or other follow-up actions can be triggered.

Software AG offers a platform for Intelligent Business Operations (IBO) [Ope14] which already includes parts of the proposed stage tracking functionality for business processes.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces stages and milestones for business processes. Section 4 gives an overview of our solution, while Section 5 presents its implementation. Section 6 addresses future extensions of our approach. Finally, we conclude with a summary in Section 7.

2 Related Work

Some IT vendors offer BPMS functionality that relates to some extent to the notion of milestones and stages for business processes.

With Business Monitor IBM provides a tool to monitor business process instances, including their notion of milestones. A milestone is a collection of so-called monitoring contexts. These milestones are used as single steps in a virtual process diagram. The user can specify under which condition a milestone gets a different fill color or labeling in a corresponding visual cockpit.

Oracle Business Process Management uses the notion of Guided Business Processes to organize process activities in milestones. This approach concentrates on interactive activities in a business process. The notion of milestones is focused on business processes involving human interaction. Milestones can be equipped with expiration dates or times and subsequent actions are triggered when milestones are breached.

The Appian BPM Suite for Designers provides capabilities for Intelligent BPMS Analytics as well as Complex Event Processing. Intelligent BPMS Analytics combines particularly BAM and Predictive Analytics. Business processes can be monitored using real-time performance metrics. With the help of predictive analytics this real-time information can be used to predict the process behavior.

SAP also provides capabilities to monitor milestones in a process context. The monitoring process can use rules to decide whether a relevant exception situation has occurred.

[MFE12] discusses CEP- and statistics-based monitoring for the transport and logistics domain, more precisely the standardized Cargo 2000 process. While this process contains fixed milestones and seems to be linear, our approach works for arbitrary business processes from arbitrary domains. Additionally we support more complex processes in-

cluding branches and iterations etc. as well as arbitrary process milestones.

[BDR12] addresses prediction and root cause analysis of events against the background of Business Activity Monitoring. Key performance indicators (KPI) and associated rules are used for the analysis. Predictions are created for rules associated with KPIs. When the associated prediction engine has finished a training phase, the predictions can be used to forecast rule violations and provide warnings. This work concentrates on KPIs and rules, while we address process stages and milestones. While databases for maintaining relevant data as well as polling intervals for the evaluation of the current status are used, we use Complex Event Processing technology in order to support real-time processing of large volumes of process data.

[CK07] proposes a model-driven approach to enhance already present business performance models with predictive modeling. So-called metamodels describe languages which can be used to express models. These metamodels are extended by a notion of time in order to allow for the prediction of metrics. Further trigger conditions are provided that define when and how the predictive analytics tasks are executed. The tasks themselves are provided as a service of a dedicated component, which gets the relevant data from a data warehouse and also persists the forecasted metrics into the warehouse. This work is a conceptual approach describing in a model-driven fashion how monitoring metrics can be extended towards forecasting and how the resulting models are generated and processed. It does not discuss notions of process stages and milestones as well as of an associated real-time monitoring based on Complex Event Processing.

[CFLZ13] discusses prediction of metrics based on events. Within a training phase selected events are stored in an intermediate storage. Then they are grouped and aggregated with summary statistics. These summary statistics are used to detect patterns within the events. A function selector then uses this information to select a suitable prediction model which is finally used to predict metrics. This approach delivers a framework for un-supervised prediction using events. The specific problem of monitoring stages and milestones of a business process and raising early warnings in case of potential stage violations is not addressed.

[BBH+14] addresses the monitoring of business processes with Complex Event Processing. A general framework is presented that describes how a business process environment can be extended by a Complex Event Processing engine so that business processes can be monitored. The focus of this paper is the discussion of the different components of such a framework and the basic interaction patterns.

[WZM+11] discusses an event-based monitoring of process execution violations. While we focus on violations of stages, these execution violations refer to deviations from the control-flow during execution of a process, i.e., is the current execution sequence compliant with the associated normative process model.

In comparison to these related approaches, our novel approach combines the following benefits: Our framework provides the business user means to monitor business processes in a very fine-granular manner as it allows him to concentrate on the crucial parts of the process and define meaningful monitoring criteria for these parts. This is encapsulated within the concept of stages and milestones being tracked. Additionally the user is alerted

proactively in case the process parts being monitored run the risk of failing the specified criteria. In order to provide maximum transparency to the user, the framework is based on EDA and CEP principles. This approach guarantees instantaneous results as well as scalability for the case of massive data volumes.

3 Preliminaries

In the subsequent discussions we use the following running example: An insurance company uses business processes to handle the processing of incoming insurance claims. The execution of such a process consists of automated steps, like sending the client a rejection or triggering the loss payment, as well as of manual steps requiring human interaction, like reviewing material documenting the damage or approving a payment estimate. Similar process steps can also be found in many other domains, e.g. handling calls in a call center or processing credit applications.

Such business processes are to be monitored by tracking associated milestones/stages. For this business process a process model is available that formally defines the tasks and their execution orders. The business process is executed by one or more process instances in a process execution environment. Within that execution environment sensors are used to monitor the process instances so that changes of a process instance are captured and directly populated to follow-up consumers for further analysis. For that purpose an Event-Driven Architecture (EDA) can be leveraged. Each time a process instance change takes place, a corresponding process instance change event is built and published. Such an event carries the information that a process instance change has taken place at a given point in time, e.g., the task 'Approve claim' of process instance 4711 has been completed at 14:23. Thus, during process execution all running process instances continuously emit information about their latest changes. For publishing and consuming those events the Event Bus within the EDA is used. That Event Bus allows multiple participants to publish events on event channels and also consume events from event channels they have subscribed to.

The process instance change events can be used to monitor running process instances by evaluating the status of stage and milestone completion. A milestone refers to one specific position within a process, e.g., the task 'Review damage documentation' has been started. A stage consists of a start and an end milestone and a stage condition. The stage defines that the steps between completion of start and end milestone have to meet the stage condition. A common class of stage conditions is of temporal nature. For example, the timespan between start of the task 'Review damage documentation' and start of the task 'Request additional information from client' has to be less than one day. Other stage conditions refer for example to the number of iterations of process steps.

Using the aforementioned process instance change events, which are continuously emitted, the stages can be continuously tracked. Such a tracking requires sending, analyzing, and receiving of corresponding process events in a real-time fashion.

4 Overview of Stage Tracking Framework

This overview introduces a framework for tracking stages of running business process instances, followed by a detailed discussion of the implementation in the next section. The framework comprises the following layers and components:

Stage configuration layer: On that layer the user can configure process stages for a given business process model. This includes defining the start and end milestone of the stage as well as the associated stage condition. The user can also define whether potential stage violations are to be checked. If that functionality is enabled, the user can optionally provide input for the initial setup of the statistical models used to determine stage violations. Once the stage plus its monitoring options has been defined, it can be uploaded for execution, i.e., the stage is actively monitored for currently running process instances.

Process data acquisition layer: On that layer the relevant data for evaluating stage completion status as well as potential stage violations is acquired. The process engine executing the process instances is equipped with sensors. These sensors trigger a process instance change event each time a process instance changes its status. These streams of process runtime events are directly forwarded to subsequent consumers using the EDA.

Stage analysis layer: That layer hosts the main logic to determine stage completion status and potential stage violations. While process instance change events are streaming in, the completion of stages is continuously monitored using a Complex Event Processing engine. For each incoming event all stages defined for the associated process model are determined. For each of those stages it can be derived when the milestones of a stage have been completed. That information is then used to check whether the stage condition is fulfilled. Depending on the result corresponding result events are published that indicate whether the stage has been breached or completed. Additionally the stage analysis layer determines potential stage violations. For that purpose it maintains for each process model statistical models of the process steps involved. Such a statistical model models the cycle time of a process step based on knowledge from previous and current process instance executions. Additionally the analysis layer is connected to the process data acquisition layer and thus receives the streams of process instance change events in a real-time fashion. Each time a new process instance change event arrives, the statistical model of the associated process step is updated. Given the current step of the process and the process model, all potential paths from that step to the end milestone are determined. For each of those paths the probability that the time from current step to end milestone takes longer than the remaining time to expected stage completion is determined. For the computation of that probability the statistical models are exploited. Then the computed probability for potential stage violations is forwarded as result event for further consumption.

Result processing layer: The result processing layer continuously receives events from the stage analysis layer describing completion status of stages as well as potential stage violations. These events include details like the process instance id, the stage id, and the violation probability. This information is combined with the process model and process execution in a visual cockpit so that the user can quickly assess the impact of stage completions and potential violations on his current business.

The following figure 1 shows an example stage that has been successfully completed. Evaluating the completion of the start milestone and the completion of the end milestone reveals that the stage condition has been met and thus that the stage has been successfully completed for that process instance.

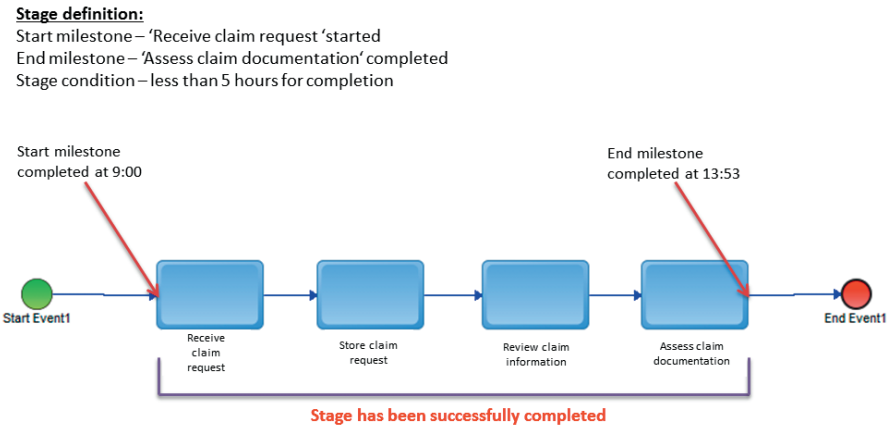


Figure 1: Example for successful completion of a stage

The following figure 2 illustrates the main idea of stage violation monitoring with an example. Using the estimated time till completion of the end milestone and comparing it to the time left for successful stage completion reveals that in this case the risk for a stage violation is very high.

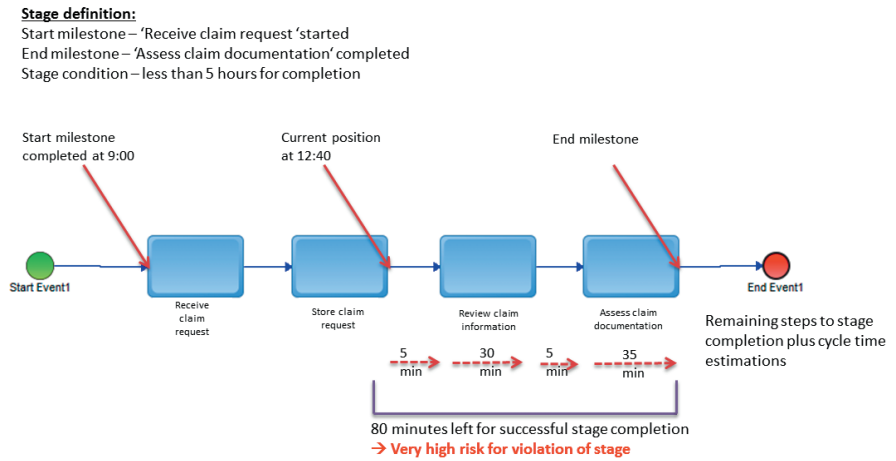


Figure 2: Example for potential violation of a stage

5 Implementation of Stage Tracking Framework

Based on the previous overview this section discusses how the corresponding layers can be implemented. Software AG offers in its platform for Intelligent Business Operations already the tracking of stages and milestones.

5.1 Stage Configuration Layer

The user utilizes the stage configuration layer to define and configure process stages for processes. As already described the user can select a process model and define start and end milestone of a stage as well as the associated stage condition. Next he can activate the monitoring of potential stage violations. This monitoring component relies on statistical models of the process steps, which describe their cycle times. More precisely, such a statistical model consists of a probability density function which describes the distribution of the cycle time. These models can either be learned automatically using process runtime data or can be specified by the user, thereby leveraging the expert knowledge of the user. For the latter option the configuration layer offers different parameterized statistical models. The user selects the model that fits best and then specifies the corresponding parameters of that model. Examples for such parameterized models are:

- Discrete single point distribution: Only one value has to be provided, which has a probability of 100% to occur.
- Discrete multiple point distribution: A set of values has to be provided, each value having assigned a probability. A special case is the discrete uniform distribution where each of the possible values has the same probability.
- Continuous uniform distribution: This distribution consists of two values, which are minimum and maximum. Between minimum and maximum the probability is uniform and 0 otherwise.
- Normal distribution: The normal distribution is identified by mean and variance.

If the user enables automatic learning of the statistical models, the models are learned using process runtime data. If the process instances have been freshly started, these models are not yet available. Therefore the user can either select to wait or to specify and use intermediate models until the learned models are available. For these intermediate models, again, the user can select from a list of parameterized models and set the corresponding parameter values.

Additionally the user can define different visualization options and follow-up actions depending on the severity of the stage violation. For example, if the probability is above 90%, the corresponding visual cockpit highlights that stage, e.g., by filling it with color red, and additionally an email is sent or a follow-up process is triggered. If the probability is below 30%, the stage is marked as green.

As a final step the process models, the defined stages as well as the configured monitoring options are uploaded for execution.

In the Software AG suite the Process Designer is used to specify process models and stages/milestones. Within that modeling tool the user can build and upload a process model for execution once the modeling and configuration of the model has been completed. During that upload step the Process Engine transforms the incoming process models into executable processes. Additionally the stage definitions are written to a database.

Figure 3 summarizes the main steps for the configuration of stage violation monitoring.

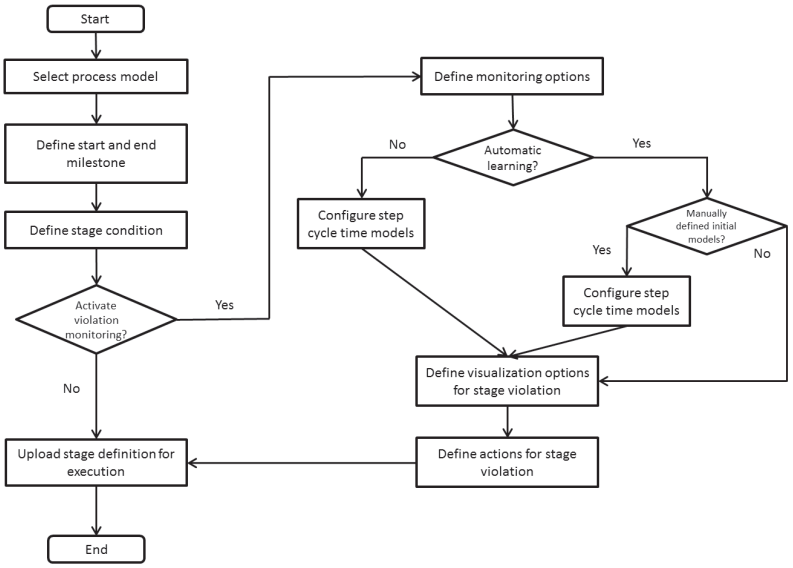


Figure 3: Main steps for configuration of stages

5.2 Process Data Acquisition Layer

Process Engine executes the process instances for uploaded process models. In the execution environment of these process instances internal triggers are registered, which listen for process initializations and step transitions. The registration of these internal triggers is done automatically; the user does not need to configure how and where process instance information is acquired during execution. No additional steps are required other than those already done in the process modeling step; Process Engine configures the internal triggers within the model upload and execution step. During runtime Process Engine reports each status change of a process instance and generates an internal document with detail information like process instance id, process model id, step id, point in time when the change took place etc. Each process model has a corresponding trigger associated, which listens to those generated documents with execution details of instances of that model. Therefore

the corresponding trigger executes every time Process Engine reports a status change of a process instance. This approach allows for a fine-granular monitoring of a process model as every transition of all process instances is reported.

After a trigger has received such an internal document describing a process instance change, that information is extracted and encapsulated into a new process instance change event. This event is to be published so that follow-up consumers, e.g., a visual cockpit or the aforementioned process tracking component, can be informed about recent changes of process instances. For that reason Process Engine is EDA-enabled, i.e., it is connected to the Event Bus, which is the general transport layer for arbitrary events in the Event-Driven Architecture. Process Engine can therefore be publisher as well as consumer within the EDA. In this context, Process Engine directly emits the newly built process instance change event to a dedicated channel of the Event Bus. Over that channel all process instance change events are published, i.e., that design also allows to capture information of multiple process engines. A corresponding tracking component subscribes to that channel and gets the events in an event-driven manner. This approach ensures a minimum time between change of a process instance and analysis of that information with respect to affected stages.

The results provided by the stage tracking component are processed in the same manner. Directly when new insights have been derived, a new result event is built and published on the Event Bus so that subsequent consumers like visual cockpits can process that information directly. Figure 4 summarizes the main flow of events.

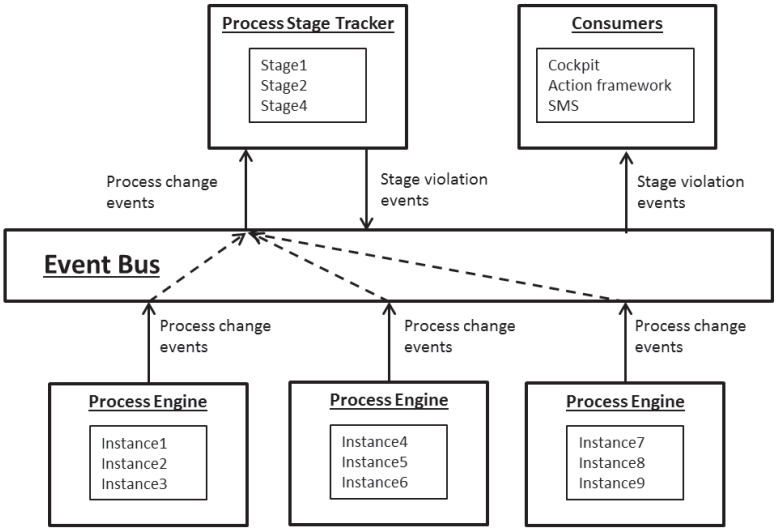


Figure 4: Event flow

Enabling such a monitoring of process instances for a process engine introduces some overhead as process instance change information has to be captured and published as an

event. That overhead will be manageable in most cases as the Event Bus bears the main burden of processing and publishing these events to follow-up consumers and the CEP engine bears the burden of analyzing the events. However, in cases where huge numbers of process instances change with a high frequency, the additional step of generating and sending process instance change events can adversely affect the performance of the process engine. Such a problem can be alleviated by distributing the process execution load over multiple process engines.

5.3 Stage Analysis Layer

The stage analysis layer is responsible for receiving process change events, determining status and potential violations of registered stages, and publishing the results to follow-up consumers. The main components of that layer are summarized in Figure 5.

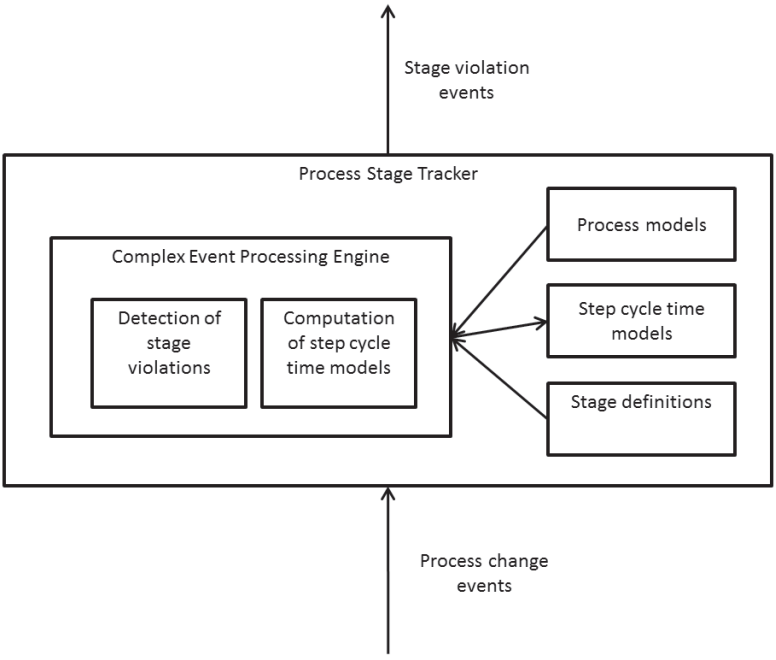


Figure 5: Main components of stage analysis layer

5.3.1 Processing of Incoming Events

The process stage tracker continuously receives process instance change events from one or more process execution engines. A process instance change event denotes for a particular process instance that either a process step has changed or a process transition has taken

place. A process step can change its status from started to completed. Besides started and completed more complex cases are possible like expired, waiting, interrupted etc. In the course of this work we focus on the case of status started changing to status completed. A process transition is the transition from one step being completed to the start of the subsequent step.

For each those incoming events the following actions are executed: First, the event is used to determine stages being registered for the associated process model. More precisely, the event is joined with the database that stores the stage definitions. Second, the stage status is derived. For each of those stages, it is checked whether start or end milestone of that stage is completed. If the latter is the case, the stage condition is checked. If the condition has been met or failed, a result event is published on a dedicated event channel. This functionality is implemented by means of Complex Event Processing queries to ensure a minimum processing latency. For example, a continuous query joins the incoming events against the stages database. The third step is to detect potential stage violations and publish corresponding stage violation events. The process instance change event is utilized to update the statistical model for the associated process step or transition. Combining these statistical models of the process cycle times for the different process steps and transitions, the probability for failing the stage condition can be computed, as will be discussed next.

5.3.2 Statistical Modeling of Cycle Times

Given a process model a statistical model for the cycle time of each step and transition is computed. This cycle time is defined as the elapsed time between start of a step/transition and completion of that step/transition. According to the previous considerations the cycle time can be computed by evaluating the streams of process transition events. For that purpose a continuously running CEP query can be used which joins the start of a process step/transition with its completion. Using the temporal information of these two events the cycle time can be derived. Thus, the query emits a stream of cycle times of steps/transitions of the running process instances.

This information is used to derive a statistical model. The cycle time of a step/transition is modeled as a univariate random variable X. Thus, X has a probability density function f which defines the distribution of the cycle time. This probability density function is unknown. Therefore, the measured cycle times are interpreted as a random sample of independent and identically distributed random variables following X. This $iid.$ sample can then be used to estimate the probability density function of X. The resulting density estimate $\hat{f}_X$ can be utilized to determine the characteristics of X. More specifically, the probability that the cycle time is in a given range $[a, b]$ can be estimated by

$$P(X \in [a, b]) \approx \int_a^b \hat{f}_X dx$$

For instance such a statistical model can be used to answer questions like:

- What is the probability that the cycle time of 'Receive claim request' is between 10 and 15 minutes?

- What is the probability that the transition from 'Store claim request' to 'Review claim information' takes more than 5 hours?
- What is the probability that 'Approve claim' completes in less than 2 hours?

5.3.3 Estimation of Probability Density Function

For the estimation of the probability density function using a sample mathematical statistics offers various techniques. Most prominent categories are parametric and nonparametric techniques. Parametric techniques assume that the density belongs to a class of parametrized density functions. Then the sample is used to estimate these parameters. A commonly used parametric density estimate assumes a normal distribution, which is parametrized by mean and standard deviation. Thus, the sample is used to estimate those two parameters. Nonparametric techniques make no assumption on the underlying distribution. A commonly used nonparametric density estimation technique are Kernel Density Estimators [Sil86].

Given the stream of cycle time events the density estimates for the different steps and transitions of the processes are continuously computed. For that purpose CEP queries are used. For example, if the statistical model is based on the normal distribution, the CEP query computes mean and standard deviation of the cycle times. These values are then used to derive the density estimate. If the statistical model is based on Kernel Density Estimators, the CEP query has to maintain the Kernel centers and the bandwidth. In both cases the queries refer to a sliding time window, i.e., only the events in the current time window are used to compute the density estimates. This approach ensures that the estimates focus on recent trends in the development of the cycle times. Additionally, it confines the memory required for maintaining internal states. For the case of very large windows, e.g. weeks or months, Cluster Kernels [HS08] can be used for kernel density estimation as they allow the online computation of density estimates over streaming data. As mean and standard deviation require for an online computation only a constant amount of memory, also the normal distribution can be computed for very large windows.

Overall, the CEP queries continuously derive statistical models of the associated cycle time distributions from the stream of measured step and transition cycle times. The resulting statistical models for each transition and step are maintained in a model database/cache. Besides the latest model, that layer also can store histories of the models in order to allow for comparisons between recent and past behavior. This history is particularly relevant for storing significant changes in the cycle time distributions which occur only seldom or periodically. For example, in summer season the number of insurance claims related to lightning strokes is typically higher. These claims are more difficult to process as the payment amount is typically high, on-site investigations of the damages are necessary, etc. Therefore the cycle times for processing these claims are higher.

5.3.4 Statistical Models for Two Process Changes

So far we have only discussed the cycle time distribution for a single step/transition. Additionally, a model is required for a sequence of subsequent steps and transitions. Given a step and a subsequent transition, the question is how long the combined cycle time for those two is. According to the previous discussion we have for both a separate statistical model. Let X_1 be the random variable underlying the step cycle time and X_2 the random variable underlying the transition cycle time. Then the combined cycle time can be modeled as the distribution of the sum of X_1 and X_2. Given the densities for X_1 and X_2 the density of that sum is defined as

$$\hat{f}_{X_1+X_2}(x) = \int_{-\infty}^{\infty} \hat{f}_1(y)\hat{f}_2(x-y)dy$$

Thus, the density estimates for X_1 and X_2 can be used to derive an estimate for the sum of those two random variables. Regarding evaluation of that formula, in some cases this integral can be resolved with a closed formula, e.g., for the sum of two normal distributed variables. If that is not the case, numerical integration techniques can be used for an approximate closed formula.

Thus, this approach can be utilized to derive a statistical model for the cycle time of a step followed by a transition. The same holds for a transition followed by a step.

5.3.5 Statistical Models for Multiple Process Changes

Next case is that multiple transitions, not only one, end in one step. In that case the maximum of the transition cycle time distributions is considered first. The statistical model for that maximum distribution can be derived by using the product of the density estimates of the contributing transition cycle times as overall density estimate. That density estimate for the transitions is combined with the density estimate for the subsequent step. The overall cycle time is the sum of the maximum cycle time of the transitions plus the cycle time of the step. Again the two density estimates for the transitions and the step are combined with the above formula to derive a density estimate for the overall cycle time of multiple transitions ending in one step.

Overall, these mechanisms allow to estimate the cycle time of two subsequent process changes. This can be generalized so that the cycle time for more than two subsequent process changes can be estimated. This in turn provides the means to estimate a process stage violation. Given two arbitrary positions A and B in a process, the models can be used to determine the probability that getting from A to B requires a certain amount of time, e.g., the probability that the time from A to B is between 15 and 20 minutes is 73%. Figure 6 illustrates that approach for the cycle time from 'Receive claim request' being completed to 'Assess claim documentation' being started.

The database/cache for the statistical models has an additional access layer for the latest statistical models. It basically offers the following query functionality:

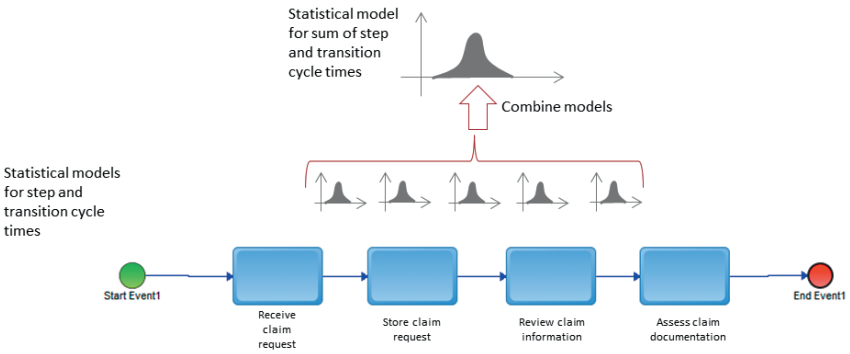


Figure 6: Computation of combined statistical model

- Input: A process model id, two positions A and B in the process, a time interval $[a, b]$
- Output: All paths from A to B plus the probability for each path that the time for the path traversal is in the time interval $[a, b]$

Regarding the actual computation of the statistical models while process instances are being executed, different phases are distinguished. When the process instances are initially started, the learning of the step cycle time models using measured cycle times as previously described requires some time. Therefore as discussed in the configuration chapter the user can specify initial statistical models based on his expert knowledge. In the initial learning phase these models are used. Once the statistical models have been learned, the system starts to use them instead of the user-specified models, provided the user has activated automatic model learning. For the learned statistical models different statistical techniques have been presented to determine the corresponding density estimates. By default a normal distribution is assumed for the following three reasons: First, the required parameters mean and standard deviation can be efficiently computed by means of simple Complex Event Processing queries using aggregates. Second, the parameters can be stored with minimum effort. Third, the sum of these statistical models can be computed with a closed formula.

Even though a considerable amount of real-world phenomena can be suitably modeled with a normal distribution, in many cases it does not fit. For example, the insurance company receives a significantly higher number of insurance claims due to a recent hailstorm. To deal with that overload also teams so far not familiar with that class of claims are assigned to these claims. Most likely, these team members will need more time for processing a claim than members familiar with those claims. So the cycle times of corresponding process steps more likely have a bimodal instead of a unimodal distribution. Similar effects can arise after sending some team members to trainings to improve their skills in handling more complex claims. In order to deal with these cases and to prevent decisions being made on the basis of an inadequate statistical model, the suitability of this normal distribution assumption is periodically checked by the system. A statistical test, e.g. the

Kolmogorov-Smirnov-Test, is used to test whether the measured cycle times, on which the model is based, are normal distributed. If that is not the case, the system switches to a nonparametric statistical model, which comes at the expense of higher complexity for model computation and combination of models, but fits better to the actual data.

5.3.6 Detection of Potential Stage Violations

The continuously computed statistical models are leveraged to detect potential stage violations. For each incoming process instance change event it is checked whether associated stages may be violated. Thus, for an incoming event the first task is to retrieve all stages that are registered for the associated process model. From these stages only those are selected where the current process state lies between start and end milestone. For each of those stages the start and end milestone as well as the associated temporal condition are determined. The process instance change event gives information on the current step or transition as well as when the change took place. This point in time is combined with the stage information to determine the remaining time to reach the end milestone. This requires additionally the time when the start milestone has been completed. Given the start milestone completion time, the current time of the event, and the temporal condition, the remaining time to stage completion can be derived. In the example in Figure 2 the stage definition requires that the time from the start of 'Receive claim request' to the completion of 'Assess claim documentation' is less than 5 hours. The start milestone, namely 'Receive claim request' has been completed, was reached at 9:00. The current event states that 'Store claim request' has been started at 12:40. Thus, the end milestone, completion of 'Assess claim documentation', has to be reached within 80 minutes in order to not violate the stage. The remaining time to end milestone completion as well as the current process state are then combined with the statistical models. As described in the previous section the probability that the time from current process state to end milestone completion is less than the remaining stage completion time is computed. This delivers the probability for violating that stage. For example, a process instance change event stating that process instance 4711 has completed 'Store claim request' at 12:40 may result in two events stating that Stage1 is violated with 70% probability and Stage2 with 17% probability.

Up to now stages with 'less than' conditions have been described, i.e., the stage condition demands a completion in less then a specified timespan. Analogously the same approach can be used to detect violations of 'greater than' stages, i.e., the stages have to take at minimum the specified amount of time.

Regarding the implementation of that functionality, again CEP queries are used. Each time a new process instance change event arrives, events describing potential stage violations are computed and directly published to follow-up consumers. Such a stage violation event provides information about

- Process model
- Process instance
- Current process step

- Stage
- List of paths to the end milestone
  - Steps of each path
  - Violation probability for each path

5.3.7 Result Processing Layer

The final layer processes the results produced by the stage analysis layer. It receives continuously events describing stage status as well as potential violations. These results are presented to the user in a suitable UI. This UI is real-time enabled, i.e., each time a new violation event is received, the corresponding components of the UI are directly updated. This is implemented by means of the underlying Event-Driven Architecture.

There are different ways to visualize stage violations. Some of them are exemplary discussed. A table can visualize all recently received stage violations. In order to limit the size of that table, events can be discarded depending on the severity of the violation, e.g., all violations with probability below 20% are discarded. Additionally that table can sort and highlight stage violations with respect to their severity. A stage violation event may comprise different paths to the end milestone with each path having a violation probability. In that case the maximum of those probabilities is used to determine the overall severity of the stage violation. The user can select an entry in the table to get more details. The complete information of that stage violation is then displayed, including in particular the paths to the end milestone. Additionally the process model is visualized. If the user selects a path to the end milestone, this path is highlighted in the process model, i.e., the current process position is highlighted and the subsequent steps to the end milestone. For each of those steps and transitions the user can also examine the associated statistical models to gain more insights into the actual behavior of the process steps.

The following example screenshot in figure 7 combines these components. The user selects a stage violation and gets details on the selected stage, the different paths to the end milestone, the associated process model with the selected path being highlighted, and the statistical models for each step on that path.

Besides visualizing the stage violations other actions can be triggered for an incoming event. For example, a follow-up process is triggered, a call is executed, or a SMS is sent. Again, these actions can be triggered dependent on the severity of the stage violation.

6 Extensions

The presented framework for tracking important sections of business processes can be extended in different areas:

Inclusion of intra-process dependencies in cycle time models: In the previous discussions on modeling the cycle times a separate model has been established for each step,

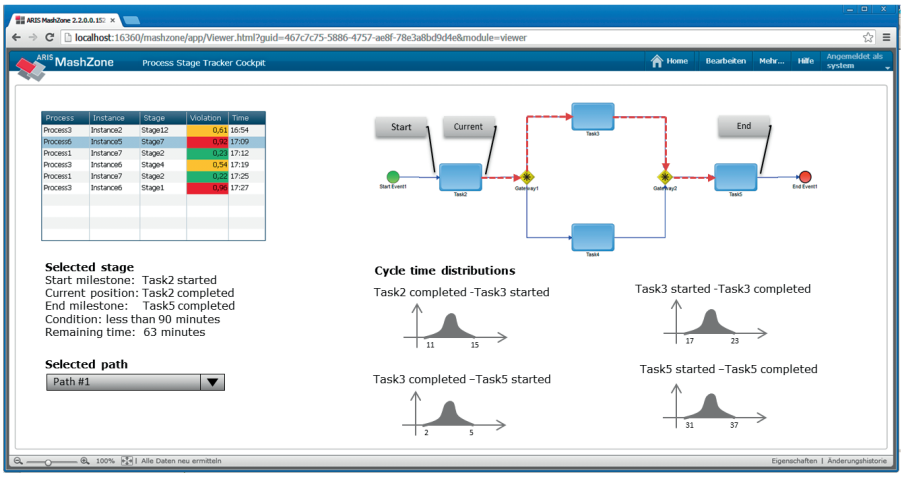


Figure 7: Example for a Process Stage Tracker Cockpit

with the assumption that the cycle time of one step does not interfere with the cycle time of the previous step. If we skip this independence assumption, a conditional cycle time model has to be used that copes with the dependencies of step executions.

Inclusion of inter-process dependencies in cycle time models: So far we have discussed cycle time distributions for a process step being estimated using data from instances of the corresponding process. A more complex model would include also the dependencies between different processes and their effects on the cycle time distributions. These inter-process dependencies correlate the information of process instances from multiple processes.

Notification of exceptionally long cycle times: The statistical model of a step cycle time provides a well-defined estimate of the distribution. This model can be used to detect exceptionally long cycle times. By comparing a measured cycle time with the latest model, the probability for the occurrence of that cycle time can be derived. If this probability is very low, that value can be marked as outlier. The user is then informed about that outlier as it affects the estimation of the remaining time to end milestone, which will most likely be higher due to that outlier.

Other tracking targets: Besides using stages associated with temporal conditions, other types of stages can be examined in a similar fashion. These stages could for example be based on the number of process step iterations. Besides the case of process steps changing from status started to status completed, the framework can also be adapted to deal with more complex statuses like waiting, interrupted, expired etc. Also other important exceptions in the execution of process instances can be implemented within the presented framework, e.g., detection of out-of-sequence events, process timeouts.

7 Conclusions

This work discusses an extension of business process management and execution with respect to a fine-granular real-time monitoring of crucial parts of business processes. A novel framework for tracking stages of business processes in an event-driven manner is introduced. This framework allows the business user to track in real-time whether constraints associated with those stages are fulfilled or violated. The framework relies on an Event-Driven Architecture (EDA) coupled with a Complex Event Processing (CEP) engine. The process engine executing the business process instances emits process status events within the EDA. The CEP engine receives those events and uses queries to determine whether registered stage constraints are met or not. Additionally, statistical models of the cycle times of process steps and transitions are continuously determined. These models are used to derive the probability that a stage constraint is violated, i.e., that the successful completion of a stage is at risk. This combined stage tracking functionality delivers the business user real-time insights into the execution of his business processes, with a focus on the most important sections so that he can directly or even proactively act in case of problems.

In our future work we will continue to incorporate more sophisticated business process monitoring functionality within the Intelligent Business Operations platform of Software AG. We will investigate the effectiveness and efficiency as well as the scalability of our approach in different application domains consisting of a multitude of concurrently running process instances. In the course of this investigation we will also assess the application of our process tracking solution within the context of iPRODICT [iPR14], an ongoing German research project which tackles a forecast-based adaptation of business processes in real-time.

References

[BBH+14] Susanne Bülow, Michael Backmann, Nico Herzberg, Thomas Hille, Andreas Meyer, Benjamin Ulm, Tsun Yin Wong, and Mathias Weske. Monitoring of Business Processes with Complex Event Processing. In Niels Lohmann, Minseok Song, and Petia Wohed, editors, *Business Process Management Workshops*, volume 171 of *Lecture Notes in Business Information Processing*, pages 277–290. Springer International Publishing, 2014.

[BDR12] Geoff Bullen, Richard M. Dickson, and Warren C. Roberts. Systems and/or methods for prediction and/or root cause analysis of events based on business activity monitoring related data. Patent US8140454 B2, 2012.

[CFLZ13] Hung-Yang Chang, Joachim H. Frank, Christoph Lingenfelder, and Liangzhao Zeng. Non-intrusive event-driven prediction. Patent US8468107 B2, 2013.

[CK07] Pawan Chowdhary and Shubir Kapoor. System and method for applying predictive metric analysis for a business monitoring subsystem. Patent US20070244738 A1, 2007.

[EN10] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[HS08] Christoph Heinz and Bernhard Seeger. Cluster Kernels: Resource-Aware Kernel Density Estimators over Streaming Data. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):880–893, 2008.

[iPR14] iPRODICT. http://www.software-cluster.org/de/forschung/projekte/partnerprojekte/iprodict, 2014.

[MFE12] Andreas Metzger, Rod Franklin, and Yagil Engel. Predictive Monitoring of Heterogeneous Service-Oriented Business Networks: The Transport and Logistics Case. In Ralph Badinelli, Freimut Bodendorf, Simon Towers, Sharad Singhal, and Manish Gupta, editors, *Proceedings of the 2012 Annual SRII Global Conference*, pages 313–322. IEEE Computer Society, 2012.

[Ope14] Software AG Intelligent Business Operations. http://www.softwareag.com/corporate/products/bigdata/ibo/overview/, 2014.

[SC09] Roy Schulte and Mani Chandy. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill Osborne Media, 2009.

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

[WZM+11] Matthias Weidlich, Holger Ziekow, Jan Mendling, Oliver Günther, Mathias Weske, and Nirmit Desai. Event-Based Monitoring of Process Execution Violations. In Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, editors, *Business Process Management*, volume 6896 of *Lecture Notes in Computer Science*, pages 182–198. Springer Berlin Heidelberg, 2011.

A Smart Link Infrastructure for Integrating and Analyzing Process Data

Eric Peukert
SAP SE
Chemnitzer Str. 48
01187 Dresden, Germany
eric.peukert@sap.com

Christian Wartner
InfAI e.V.
Hainstraße 11
04109 Leipzig, Germany
wartner@informatik.uni-leipzig.de

Erhard Rahm
Abteilung Datenbanken - Institut für Informatik
Universität Leipzig
rahm@informatik.uni-leipzig.de

Abstract: An often forgotten asset of many companies is internal process data. From the everyday processes that run within companies, huge amounts of such data is collected within different software systems. However, the value of analyzing this data holistically is often not exploited. This is mainly due to the inherent heterogeneity of the different data sources and the missing flexibility of existing approaches to integrate additional sources in an ad-hoc fashion.

In this paper the Smart Link Infrastructure is introduced. It offers tools that enable data integration and linking to support a holistic analysis of process data. The infrastructure consists of easy to use components for matching schemas, linking entities, storing entities as a graph and executing pattern queries on top of the integrated data. We showcase the value of the presented integration approach with two real world use-cases, one based on knowledge management in manufacturing from the LinkedDesign project and another one based on an agile software development process.

1 Introduction

An often neglected, highly valuable asset of many companies is internal process data. Huge amounts of such data is created while people work along predefined, sometimes implicit processes that are supported by heterogeneous software tools. Examples of such processes are software development, sales activities or customer relationship management. Exemplary data sources can be emails, telephone logs, files in source version control systems, customer interaction protocols or time reports to name a few. If such sources can be properly integrated and linked, it can be used to identify shortcomings of a process and give input for optimization.

However, due to the heterogeneity of the involved tools an integration of these data sources is cumbersome. Even if a single software tool suite is used to support many process steps there is still often a need to flexibly integrate additional data sources that might help to assess the overall quality of a process. Also, processes and tools change over time so

that data source schemas, their relations and links between them suffer from continuous changes.
A number of approaches can be found in literature that simplify the creation of federated data sources [SL90, BLN86], that use schema matching [BBR11, HRO06] or that apply object matching to identify links between entities [NA11, KR10]. However, most of the existing work only focuses on individual aspects of matching and storage and often comes with the restriction of fixed schemas. First approaches try to follow a schema-flexible graph-based integration approach as it is done with BIIIG [PJRR14]. It applies a metadata-based integration of data sources to a unified metadata graph to finally construct an integrated instance graph. However, BIIIG does not provide mechanisms to identify new links at the instance level between data sources.
In this paper the Smart Link Infrastructure for graph-based data integration is introduced. It is based on a schema-flexible integration of data sources into a common information graph. Existing associations from the original data source are stored as edges in the graph. But, in many cases such associations do not exist. The infrastructure supports a ”pay as you go” approach of link generation. Links between nodes of the graph are automatically computed at the instance level based on the analysis of the contents and relations of individual nodes. Nodes in the graph can refer to people, documents, transactions or any other data object that might be generated throughout a complex process. The linking results in additional edges within the information graph which then offers new ways of interaction and analysis. Within the information graph we are able to uniformly navigate, query and analyze the existing information.
For analyzing the graph of nodes and links the infrastructure relies on so called process pattern queries. Such pattern queries are different from classical relational queries. A pattern describes a complex setting involving several types of entities together with a specific set of constraints. A pattern query helps to identify the properties of a project or process that make one project or process more successful than others. For instance, in a software development process different kinds of data entities like tasks, code, bugs and worklogs are typically collected. A flexible tool to integrate and link information created by different software systems promises unprecedented value to software companies by helping to continuously increase the efficiency of their internal development processes. A simple pattern query applied to information from a software development process, for instance, could search for requirements with high complexity that might lead to the involvement of many people or have a high likelihood for bugs.
This paper makes a number of contributions:

- The architecture of the Smart Link Infrastructure for flexible data integration and analysis is introduced together with its core data integration services for schema matching, graph storage and linking.
- The information graph and pattern queries are explained.
- Front-ends for search and analysis that apply a simple but effective drag and drop metaphor are described.
- The infrastructure has been developed within a large EU project and succesfully applied for industrial use cases. We describe two real world use cases in detail to

illustrate the versatility of the Smart Link Infrastructure.

2 Smart Link Infrastructure

2.1 Overview

The architecture of the Smart Link Infrastructure is shown in Figure 1. It consists of three layers. On the top are the user interface components. These are HTML user interfaces supporting administrative tasks, i.e. data import and linking of data records as well as graphical user interfaces for the visualization of graphs, querying and the creation of reports. Since the Smart Link Infrastructure aims at an integration with existing software systems external tools like OneNote (as used in the knowledge management processes described in chapter 3) are supported as well.

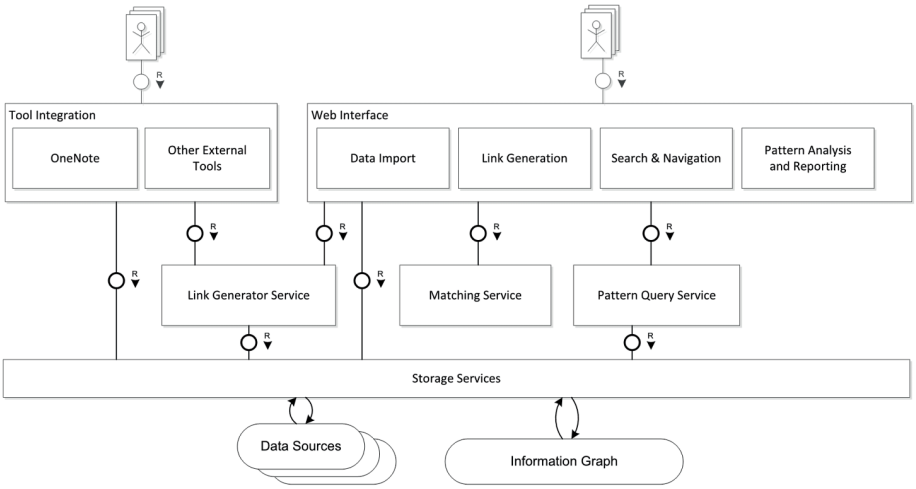


Figure 1: Architecture of the Smart Link Infrastructure

The second layer consists of a collection of services: the Schema Matching Service, the Link Generator Service and the Storage Service. The first two serve data import and management tasks while the Storage Service represents the general interface to the central graph database providing operations to store and query data. The services each expose a REST-based API towards the UI components and external clients. Communication happens largely with JSON messages.
On the bottom of the architecture the data sources can be found. One is the central graph database itself and on the left are the multitude of sources (e.g. databases, file repositories, SPARQL endpoints etc.) from which data is imported. To be able to holistically analyze

and search within data that is distributed across multiple heterogeneous data sources, various data integration approaches are used. There are some similarities to ETL processes that are used when building data warehouses to allow efficient analytic processing, e.g. when data from independently created databases is to be combined to create a unified view that gives new insights for analysts.
The general workflow applied within the Smart Link Infrastructure is depicted in Figure 2. The data integration workflow starts with the data import that is supported by a service providing schema matching methods. In this step relevant data sources can be selected like files, databases, etc. and a mapping of their schema with the schema of the Smart Link Store is created. Based on the mapping, new entities are added to the information graph. The use of the schema matching service is optional and supports administrators during the definition of the mapping. It could simplify subsequent steps if a sound input schema is available.
In the linking step, entity resolution approaches are applied to identify relations between entity sets that originate from different sources. In contrast to standard link approaches, we also support text mining techniques and new mapping types. Text mining helps to deal with unstructured data like text documents whereas new mapping types are needed to determine the type of semantic relations between entities. The linking step is important for defining a well-connected information graph, especially if the imported data sets contain no direct references to each other.
In the center of the Smart Link workflow there is a graph database, the Smart Link Store, which provides the capability to store and query data. In particular, it offers means for executing pattern queries on the information graph that are crucial for the analysis. Each step of the workflow will be detailed in the following chapters. Chapter 3 will introduce two real-world use case scenarios that utilize the Smart Link Infrastructure and give an insight into the actual usage.

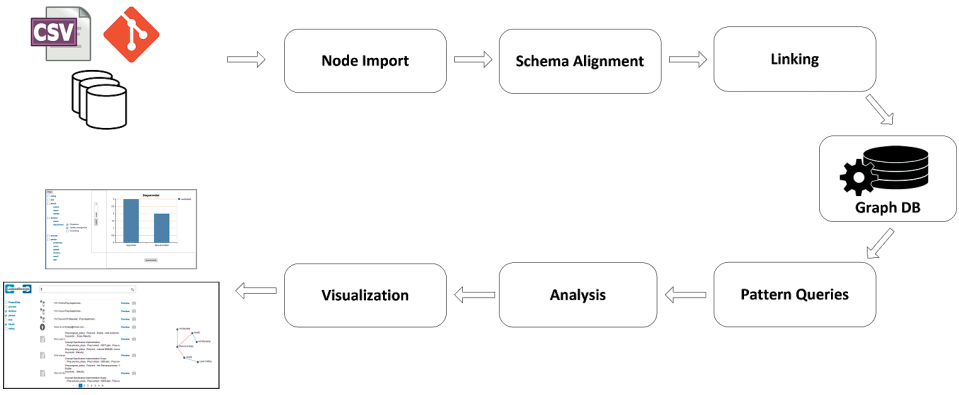


Figure 2: Overview of the Smart Link Infrastructure information flow

2.2 Data Import

In order to pre-fill the Smart-Link Store with entities, an administrative user can use the Data Import Tool. Such pre-filling allows a user to connect to existing data sources such as relational databases, Excel files or data sources that are exposed through a SPARQL endpoint. When integrating a data source into the information graph the schema of the data source needs to be mapped to the existing flexible schema of the information graph. Such a mapping describes how elements of the source schema correspond to elements of the information graph schema. As discussed above, extending the graph schema is not problematic, but existing types and properties should be reused. Adding properties to existing types or adding types is implicitly triggered when new entities are uploaded. Defining the mappings can be complex and time-consuming. It is often done manually, with the help of point and click interfaces. The data import tool of the Smart Link Infrastructure offers a point and click interface similar to existing mapping solutions (see Figure 3). Moreover,

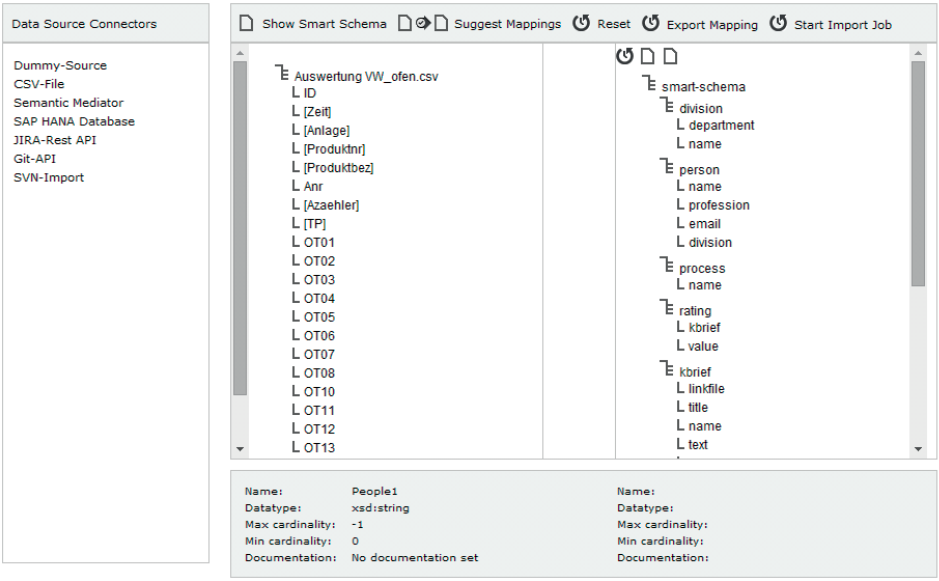


Figure 3: Point and click data import tool

a schema matching service is integrated that is able to compute suggestions for mappings.

2.3 Schema Matching Service

To reduce the manual effort in mapping, a matching service was integrated that semi-automatically computes a mapping suggestion for a user. The matching service contains

a number of matching algorithms and a library of schema importers for different schema types [PER11, DR02]. It takes as input two schemas and computes a mapping suggestion between them. Similarities between source and target elements are computed on metadata level but also on instance level. Since current matching systems are often not robust enough to be able to cope with very heterogeneous source schemas we developed an adaptive matching approach [PER12]. This approach automatically configures a schema matching system process that consists of a set of operators for matching and filtering. Based on measured features of the input schemas and intermediate results so-called matching rules can be defined. These rewrite rules rely on analyzing the input schemas and intermediate results while executing a process and rewrite the process to better fit to the problem at hand.
As was already described the adaptive schema matching approach is crucial to match the heterogeneity of schema types to an integrated schema and to finally integrate those sources within a common information graph. The schema matching service described above implements parts of the adaptive matching system and is therefore able to improve the quality of matching results.

2.4 Link Generation

Creating graphs from a structured data source like a database with well-modeled metadata can be relatively easy. In contrast to that, creating a graph from unstructured sources like document collections or independently created databases requires a component to find links between entities. The Link Generator Service allows the creation, management and execution of workflows, so called linkers, that can determine relations between entities, the relation type and the confidence of these relations. Linkers are related to the entity resolution workflows used in various data integration and data quality related scenarios(e.g. link discovery in the web of linked open data [NKH+13], web data integration [WK11] or classic ETL processes in data warehousing [CP11]). Figure 4 shows how entity resolution workflows look at an abstract level (cf. [Chr12]). Normally they are used for detecting data objects that are equivalent in the real world but have different representations in multiple data sources. The input of an entity resolution workflow are usually sets of data records from one or multiple databases. The operations in the preprocessing step include data transformation steps (e.g. to convert data types or remove special characters) and filter operations or blocking steps to reduce the search space for finding matching objects [DN09]. Match operations are then applied to determine the pairwise similarity of the candidates. Depending on the domain different distance metrics on one attribute or a combination of attributes can be used to determine this likelihood of two entities being equal [EIV07]. The match result usually is an instance-level mapping: a set of correspondences of the type (entity1, entity2, sim) where sim is the confidence of two entities being equal. In general a mapping contains correspondences of a single type, i.e. ‘sameAs’. This resulting mapping is then used to determine a set of matching objects and a set of non-matching object pairs from the input data source. Standard entity resolution workflows can be used to find objects in the graph store that are very similar and merge

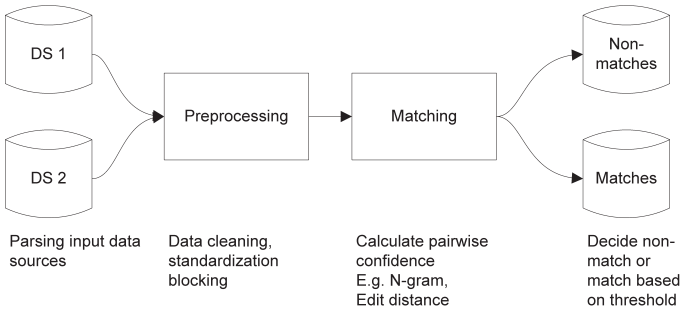


Figure 4: General entity resolution workflow

them under the assumption that these are duplicate nodes or slightly different versions of the same note. These data quality related workflows, however, are not the main focus of the Link Generator service.
Instead of searching for pairs with a high confidence of a 'sameAs' relation, linking workflows aim to search for links between arbitrary objects and their type. In the easiest case a linker can just determine a link between two objects by calculating the similarity between a certain field that exists in two objects. An example of this is a linker that tries to find documents written by the same author which involves the following steps:

1. Parse the Input (e.g. a document with a meta-data field about the author).
2. Preprocessing: Split the author field into words and sort them alphabetically.
3. Matching: Calculate the trigram similarity of the sorted author list.
4. The result is a set of correspondences between documents that in effect have the type 'sameAuthor'.

Linking workflows like this are useful to generate certain types of edges for our property graph that connect objects of the same type. However, more complex linking approaches are needed. To enable the functionality needed for the current Smart Link Infrastructure, i.e. the calculation of links between arbitrary data objects representing entities like persons, documents, etc. most of the steps depicted in Figure 4 are extended and an additional, optional, rule application step is added to the process. Figure 5 shows a general linking workflow.

Especially when trying to support search and information discovery in knowledge management processes the data often consists of unstructured text from documents with possibly incomplete metadata. The string similarity of the content of two text documents, for example, does not provide enough information about the relation between them. The only relationship that could be derived from a low lexicographic distance is that the documents have duplicate text or large parts of overlapping text. Named entity recognition, keyword detection, the detection of hyperlinks and mail addresses as well as other text mining

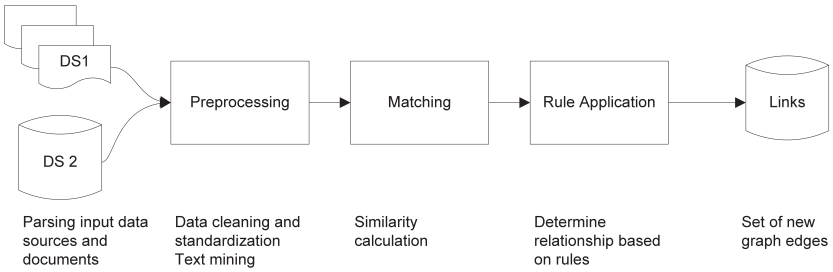


Figure 5: General linking workflow

approaches become important to link documents with semantic relations that go beyond 'sameAs' relations. A linking workflow that relies on text mining and uses a simple rule to decide the relation type could look like this:

1. Parsing of Input data(e.g. text of a PDF file).
2. Preprocessing: Named entity recognition, Keyword detection.
3. Calculation of the similarity (e.g. amount of overlap) of list of keywords, engineering concepts.
4. Application of rules: For example, a high overlap of keywords leads to the relation 'similarContent' and overlapping engineering concepts extracted from the two documents lead to the relation 'sameTopic'.
5. Links, i.e. new edges are inserted into the graph

In addition to the use of text mining approaches linking workflows can also utilize existing edges in the information graph. Since the Smart Link Infrastructure allows the incremental addition of new data sets, it is usually possible to use existing relations. An example would be a relation like 'sameTopic' which is transitive to some degree. Once a linker determines that a new and an already integrated document have a high confidence of such a relation we can infer that the certain relations exist between the new document and the already integrated documents neighborhood. This is similar to data matching approaches that use already existing mappings (e.g., [TR07]) and can be useful to discover and verify certain types of relations as well as to increase the efficiency of a linker. Generally, the workflows of the Link Generator service first produce mappings with instance-level correspondences of the type (entity1 ,entity2, linking method, confidence) representing the used linking method and its resp. confidence (e.g. (document1, document2, trigram-author, 0.8), (document1, document2, keywordoverlap-text, 0.2)). A final rule application step decides based on the applied method and the confidence value what edges are added to the graph. A set of simple conditional rules, e.g. "if confidence (author1, author2) $\geq$ threshold then linktype is sameAuthor" are satisfactory for most use cases, although their definition requires some domain knowledge.

The Link Generator service allows users the definition of suitable workflows (Linkers) for a certain domain.

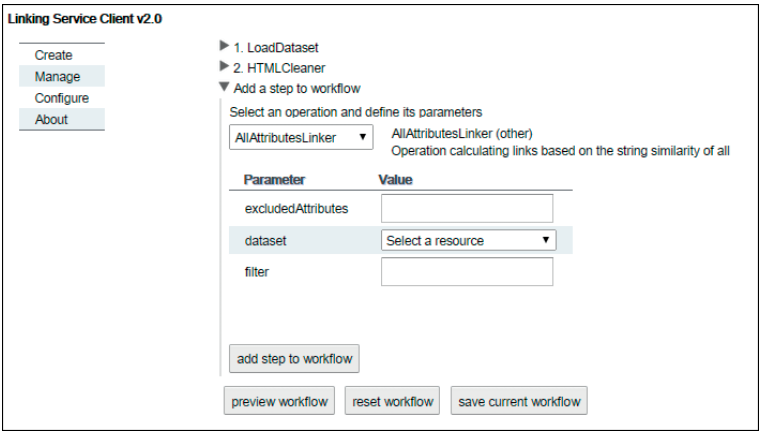


Figure 6: Linking UI

The service offers a variety of preprocessing, text mining, data matching approaches and filtering operations that can be freely combined. These include:

- Preprocessing operations, filter operations to exclude certain entities from subsequent operations,
- Keyword, link, address and named entity detection operations
- Dictionary based unification of terms
- Match approaches like N-gram distance, TF-IDF distance, Edit distance to calculate the confidence of attribute similarity
- Match approaches that determine a confidence value based on the overlap or element similarity of lists
- Match approaches based on already known relations (i.e. the neighborhood of entities)
- Combination operations for aggregating confidence values determined by different approaches
- Relationship detection rules based on individual or combined confidence values

The operation library of the Link Generator service can be extended by operations needed for a certain task at any time. Workflows can be set up through a web based UI (Figure 6) that also allows their management and setup for periodical or triggered execution.

2.5 Smart Link Store

2.5.1 Property Graph Model

The used graph store of the Smart Link Infrastructure relies on a schema-flexible property graph model with a navigational query access as implemented by Neo4J [Neo12] or AIS [RPBL13]. In contrast to widely used triple stores a property graph keeps the object identity intact. This simplifies queries since the entity construction effort can be omitted. The used property graph stores objects together with their properties. Those properties can either be atomic values or associations to other objects in the graph. Each object has a type which is managed in a dedicated type store. The set of properties of an object can be changed at any time. Also, an object type does not fix the set of types used. However, the used graph store permits to dynamically compute statistics about existing types in the store and their used properties together with usage counts. The statistics of property usage could help to filter less frequent associations to get a more stable and accurate global schema. This dynamically computed schema later helps us to generate queries.

2.5.2 Pattern Query and Query Generation

A crucial part for analyzing process data holistically are pattern queries. These are much more intuitive for a user than navigational queries since the user does not need to know the exact topology of the graph which is required for navigational queries. Navigational query access refers to the ability to traverse associations in a graph within the query language. Many graph stores only implement navigational queries such as GREMLIN in Neo4J or WIPE within SAP AIS. Existing pattern query languages include SPARQL for triple stores and Cypher for Neo4J property graphs. Due to the nature of triple stores, SPARQL queries often become rather complex. The Cypher query language for property graphs of Neo4J is much easier to use. Unfortunately it is bound to Neo4J. Typically, results to graph pattern queries are computed by applying maximum common subgraph algorithms which are expensive to execute [McG82]. Since we would like to rely on an existing internal property graph store that only provides navigational query access we decided to implement a pattern query interface on top of the navigational query interface. A pattern query is taken as input and a navigational query is generated.
Figure 7 illustrates the overall process of query generation. As described earlier, the graph model we use allows us to compute a schema dynamically from the instances that are currently in the graph. Before any query is issued to the graph store the current schema is computed. The example assumes that there is a graph with four different kinds of entities which are represented by different shapes and names of nodes. The generated schema only consists of four types of entities including possible associations between instances of these types. A user could create a pattern query by stating that there shall be two nodes directly or indirectly connected and the nodes have to follow certain filter predicates. Such predicates define which nodes from the set of instances of a type should be in the result set.
From the schema graph that was initially created, a minimal connected subgraph for the

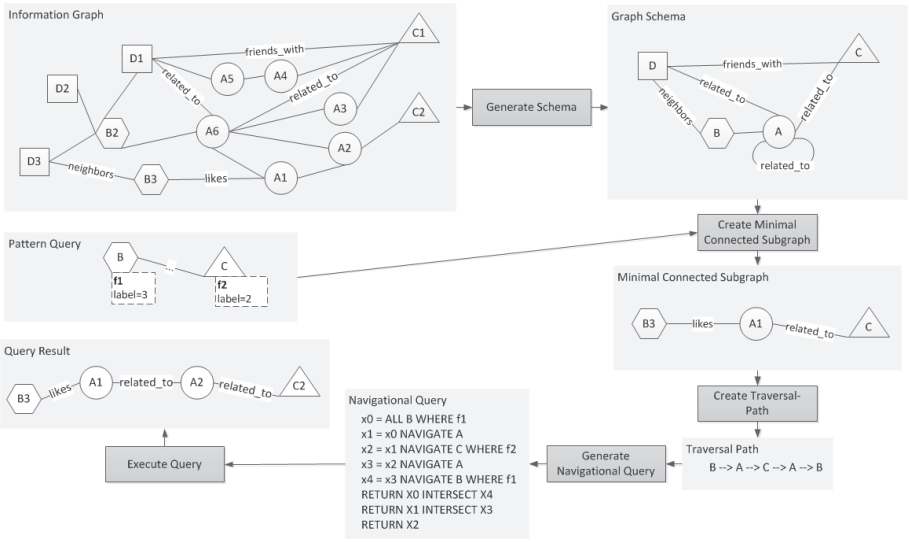


Figure 7: Pattern query generation process

given types is computed. If the types are not connected on schema level, then the query will return an empty answer. In the example, the intermediate type A was added to get a minimal connected subgraph. In order to be able to generate a navigational query on that subgraph a traversal path is needed based on the type graph. In the example the traversal starts at B and walks to C. Note that the walk needs to be done in backwards direction since initial filtering on the B nodes did not involve indirect filtering through the subset of C nodes. Consider that there are B nodes that follow the predicate f1 that have no possible path to any C node that follows predicate f2. After the traversal path has been computed, we can now generate the navigational query using any query language that allows navigating associations in both directions. That means if there is an association from A to B then the classical navigation from A to B is possible. However, in our approach we need to be able to navigate backwards from B to A returning all nodes A that have an association to B. The query language used in the example is a simplified example. At the end of the query the results of the individual navigation steps are intersected to return the correct set of nodes for each node type. In the example, nodes are returned that were originally not requested in the query. This can be filtered out depending on the requirements of the respective application.

2.6 Data Analysis

The Smart Link infrastructure offers tools for search and analysis within the integrated information graph. The information graph with its links can help to improve the ranking

(see PageRank from Google) of a key-word based search result. Moreover, the graph that can be found in the vicinity of a node is displayed in addition to the search result and entities can be filtered by facets. A screenshot of the developed search interface can be found in Figure 8.

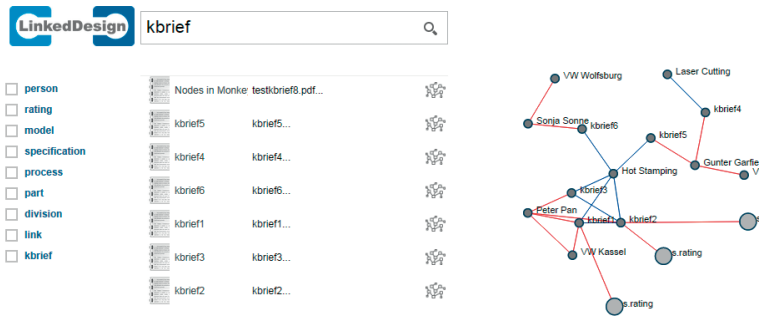


Figure 8: Search UI

The analysis of the information graph is driven by pattern queries and a simple drag and drop metaphor for creating charts. Figure 9 shows a screenshot of the analysis tool. It supports a user to graphically create pattern queries. By dragging and dropping dimensions and measures onto the axes of an empty chart, a new report can be created. In addition, users can define functions over measures to compute complex measures on top of the pattern query result.

3 Case Studies

3.1 Software Development Process

The Smart Link Infrastructure was used in a proof of concept implementation to analyze process data that is created within an agile software development process. In order to get an idea what data sources were integrated the development process is briefly described. The development is organized as an agile project based on the SCRUM method. A project is split into short sprints of 1 or 2 weeks. Requirements are collected within a so called backlog. At the beginning of a sprint, tasks (issues) are taken from the backlog, their effort is estimated and the issue is assigned to the sprint in a so called planning meeting. The development team takes issues and marks them as resolved when finished. Software code and documents are created and pushed to special code management systems. After each sprint the completed issues are reviewed and the next sprint is planned. There are many tools such as Agilefant [Agi] or JIRA [Atl] that try to support such processes. These tools offer support for issue creation, assignment to developers as well as time and status reporting. For the actual development and code management, source version control systems can

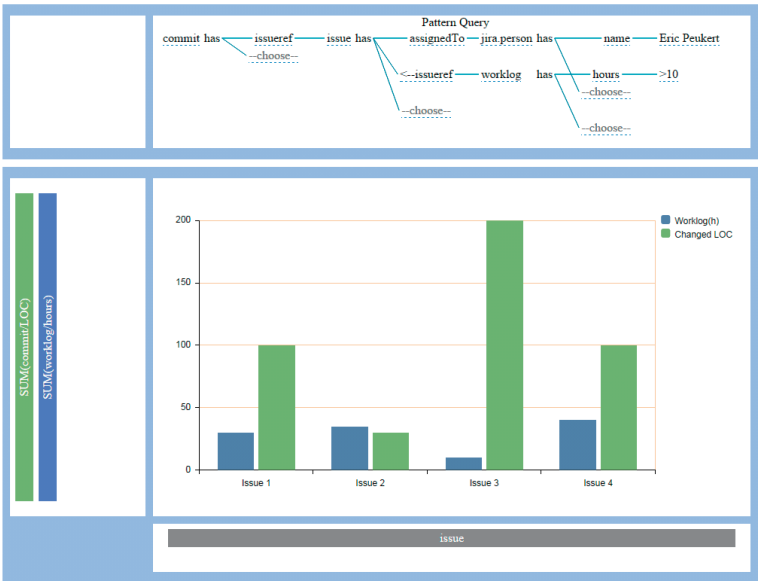


Figure 9: Analysis UI

be used such as CVS, SVN, Perforce or Git. Throughout the process, meetings are organized between team members and emails are exchanged, that contain discussions about the current development tasks. Continuous integration and test systems take developed code and run tests. Committed code is analyzed and executed which results in error/test reports that provide the developer with feedback about the quality of the committed code. In later stages, bugs are filed and attached to certain code fragments. For each of these process steps separate software systems and data stores are typically used. However, it would be valuable to be able to link tasks to actual commits, corresponding files and test reports. In the proof of concept the data sources were integrated with the help of the Smart Link Infrastructure in a common information graph that links requirements with tasks, people, code commits and test reports. (see Figure 10).

The linking service of the Smart Link Infrastructure created a number of meaningful links between different types of objects. A linking between commits and issues was possible since some developers put contents from the issue description and issue names into the commit message. Commits are also linked to people based on user ids, emails and names. Some objects are also linked by date which helps to assign commits to sprints. Issues are also linked to people which finally forms a nice information graph of the development process at hand. With the help of the analysis tools the development process could now be analyzed. It was now possible to observe interesting properties of the process that could serve as input for improving the process in future. For instance, times of high workload could easily be identified. Even though the actual estimated times for issues remained similar for all sprints the actual logged working times and committed lines of code showed

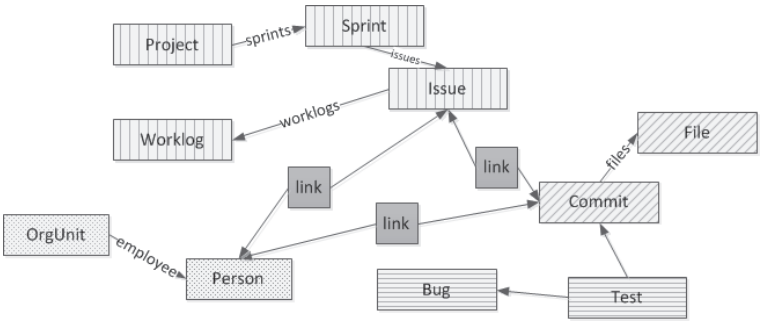


Figure 10: Linked data sources in the information graph

some strong peaks. This happened often before milestone deadlines. However, according to the scrum method, such peaks should be avoided since they could put too much pressure onto a development team. Moreover, the tools helps to measure the efficiency of a team and could give an indicator to the complexity of a software system. When a software is newly build, many lines of code are initially created in very short times. Unfortunately, the number of changed lines per logged hour of work reduces over time since the complexity of change increases. The exemplary analysis from Figure 9 showed a comparison of changed lines of code to actually logged working hours for a selected set of issues. With additional data sources like vacation times or calendar data one could certainly make further helpful observations.

In summary, the integrated information graph provided many insights on the running process that would otherwise be hidden. The lightweight integration approach of the Smart Link Infrastructure helped a lot since there was no need to define a global schema upfront. The sources could simply be integrated by letting the automatic linking identify correspondences.

3.2 Smart Link Infrastructure for Knowledge Management Processes

To demonstrate the wide spectrum of application cases the Smart Link Infrastructure was applied to analyze and support a knowledge management process within an oil platform construction company. In particular the linking component was tightly integrated into a document authoring tool (Microsoft OneNote) that is typically used within that company.

The Smart Link infrastructure is able to automatically link a document to an existing corpus of documents while it is created or edited. Found links are dynamically inserted into the currently edited document and also within related documents of the corpus. Links can be created between documents of the same kind but also to external sources such as taxonomies of terms or other records such as people, files on a share or a source version control system. This approach does not require the user to collect links to entities from each of these possible information sources manually or to manually insert them into the

document at hand. Given that these information sources are known, all such links are added automatically.

The prototype consists of an extension (Linking Plugin) to Microsoft OneNote. While editing textual content the Linking Plugin extracts the content and triggers a linking workflow of the Linking Service. The Linking Service computes links from the newly created document to existing documents or to other sources. Each linker can produce different types of links such as *references, is-related, is-contained-in, created*. As described above, the linking service can be extended by further linkers and workflows.

The linking result is filtered and ranked based on the computed confidence values. Selected links are finally inserted in the currently edited document. Links can be inserted as separate sections, but could also be added inline to the text if specific keywords relate to an entity in one of the attached information sources. Changes to the text or referenced documents can also trigger a removal of previously computed links. Generated links could be marked as being strong or less strong. Also the age of the link could be visualized. The longer a link is part of a document the more time is should take to remove the link. It should be shown as deprecated so that a user can accept removal manually. Linked documents are also changed to also link to the newly created document. The linking process is dynamic so that by writing new content the dynamic links can change. The user has the option to manually remove link suggestions or to reset linking parameters so that only links with higher confidence are added to the document.

3.2.1 Link Generation

For generating links the linking component that was described above is used. In the following we introduce a few interesting linkers. A *file reference linker* looks up file names and paths in existing file-shares to recognize file-names inside the text at hand. Also names of other documents mentioned within the text can serve as indication for creating reference links. To identify related documents text-similarity measures like TF-IDF or N-Gram are applied. These measures attach a confidence to a link that can be used for filtering and ranking. The linking quality can be improved by extracting most representative keywords from the text and to compute similarity based on comparing keyword-sets. A number of techniques propose to identify structure of documents, apply stemming to normalize and tokenize longer terms. Such preprocessing can help to improve linking quality.

The computed links between documents, authors, files and other entities build a graph of entities in the background. A neighborhood linker uses this graph to identify indirectly related documents and further information such as the authors of such documents.

3.2.2 Search and Analysis

Through the above linking process a complex graph of documents and other data entities is created.

In the proof of concept we also added facilities for users to rate created content and give feedback. Moreover the viewer tracks when certain documents where last viewed and

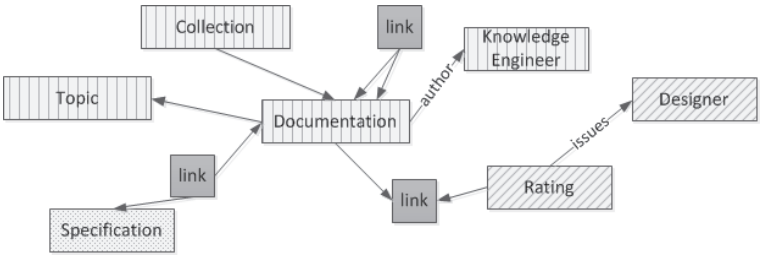


Figure 11: Document graph

edited. As a result of combining these data sources, an analysis of the age and maturity of existing knowledge documents can be performed. If legally permitted one could also measure the documentation quality and relevance of content from individual authors. The authors can be clustered by their expertise for certain topics which helps users to identify experts.
Moreover, searching over the document base improves with the help of a PageRank-like algorithm to measure the importance of linked documents.

4 Conclusions & Outlook

The paper introduced the Smart Link Infrastructure that can be used to integrate and analyze highly heterogeneous data sources that result from implicit or explicit working processes into a common information graph. Data import is supported by schema matching and links between entities of the information graph are generated by linking workflows. The integrated data of the information graph can be queried with the help of pattern queries and reports can be generated with the help of a drag and drop metaphor. Also, a search interface supports faceted search over the information graph and subgraphs can be visualized and used for navigation.
Two real world use cases proof that the Smart Link Infrastructure can be applied for solving a broad range of data integration problems that occur when integrating process data.
Future work will focus on pattern mining and query pattern recommendation techniques to simplify the construction of relevant pattern queries for a user.

5 Acknowledgement

This paper was partly funded by the European Commission through Project LinkedDesign (No. 284613 FoF-ICT-2011.7.4). LinkedDesign is a European project aiming to create a platform that addresses the problem of a missing integrated, holistic view of data across the whole product life cycle in the current manufacturing ICT landscape. The Smart Link

Infrastructure is the data integration and analysis component of the LinkedDesign platform and provides tools to integrate and link data objects from different product life-cycle phases into a property graph to enable search and analysis in order to solve problems in various use case scenarios.

References

[Agi] Agilefant. http://agilefant.com/. [last visited 10.10.2014].

[Atl] Atlassian. https://www.atlassian.com/software/jira. [last visited 10.10.2014].

[BBR11] Z. Bellahsene., A. Bonifati, and E. Rahm. *Schema Matching and Mapping*. Springer, 2011.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[Chr12] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, 24(9):1537–1555, 2012.

[CP11] Alfredo Cuzzocrea and Laura Puglisi. Record linkage in data warehousing: State-of-the-art analysis and research perspectives. In *Proc. of 22nd International Workshop on Database and Expert Systems Applications (DEXA), 2011*, pages 121–125. IEEE, 2011.

[DN09] Uwe Draisbach and Felix Naumann. A comparison and generalization of blocking and windowing algorithms for duplicate detection. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, pages 51–56, 2009.

[DR02] H. H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proc. 28th Intl. Conference on Very Large Databases (VLDB)*, pages 610–621, 2002.

[EIV07] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data Integration: The Teenage Years. In *Proc. of the 32. International Conference on Very Large Data Bases(VLDB)*, pages 9–16. VLDB Endowment, 2006.

[KR10] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[McG82] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software Practice and Experience*, 12:23–34, 1982.

[NA11] Axel-Cyrille Ngonga Ngomo and Sören Auer. LIMES: a time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of the 22. international joint conference on Artificial Intelligence*, pages 2312–2317. AAAI Press, 2011.

[Neo12] Neo4j. Neo4j - The World's Leading Graph Database, 2012.

[NKH+13] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to reach for the cloud: Using parallel hardware for link discovery. In *The Semantic Web: Semantics and Big Data*, pages 275–289. Springer, 2013.

[PER11] Eric Peukert, Julian Eberius, and Erhard Rahm. AMC - A Framework for Modelling and Comparing Matching Systems as Matching Processes. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 1304–1307, 2011.

[PER12] Eric Peukert, Julian Eberius, and Erhard Rahm. A Self-Configuring Schema Matching System. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 306–317, 2012.

[PJRR14] Andre Petermann, Martin Junghanns, Müller Robert, and E. Rahm. Graph-based Data Integration and Business Intelligence with BIIIG. In *Proc. of the 40. International Conference on Very Large Data Bases(VLDB)*, 2014.

[RPBL13] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Proc. of 15. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW)*, pages 403–420, 2013.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[TR07] Andreas Thor and Erhard Rahm. MOMA-A Mapping-based Object Matching System. In *Proc. of 3rd Conference on Innovative Data Systems Research (CIDR)*, pages 247–258, 2007.

[WK11] Christian Wartner and Sven Kitschke. PROOF: Produktmonitoring im Web. In *Proc. of 14. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW)*, pages 722–725, 2011.

Integration of Spatial Data in Database Acceleration for Analytics

Knut Stolze* Josephine Rückert* Frank Butt†

Abstract:

The IBM DB2 Analytics Accelerator (IDAA) integrates the strong OLTP capabilities of DB2 for z/OS with very fast processing of OLAP and analytical SQL workload in Netezza. Today, not all data types supported by DB2 are available for acceleration. One step to narrow this gap is to enable spatial data - in form of polygons, linestrings, and points - for OLAP processing. Since DB2 and Netezza have very different internal architectures and different limitations, a straight-forward mapping from the DB2 data type to Netezza data types is not possible.

We developed a prototype geared toward performance and simplicity for spatial data support in IDAA. In particular, all SQL statement that work against DB2 shall work against the accelerator without any changes. In this paper, we describe how the integration with the accelerator is accomplished so that ingestion and access to the spatial data becomes as seamless and transparent as possible for existing applications. The architecture and implementation aspects for spatial data representation in Netezza as well as the access during query processing are discussed in detail.

1 Introduction

Data warehouses have evolved from dedicated systems using specialized schemas like star and snowflake [Leh03] to solutions that strife for a full integration with operational data. Limiting the database schema in those ways gives a good theoretical model, but it does not always fit well with real (reporting) applications. For instance, the IBM® data warehouse industry model for banking and financial markets [IBM12] is neither a star nor snowflake schema, but surely it is intended for data warehousing. Along with such schema-based restrictions came many other requirements and limitations. For example, typical data warehouses and OLAP systems support only a subset of data types like integers, floating point numbers, and varchars. However, emerging integrated systems that target OLTP and OLAP with equally excellent performance cannot allow for such restrictions. Existing applications use data types like large objects (LOBs), XML, or spatial data.

In this paper, we present our works on extending the IBM® DB2 Analytics Accelerator (IDAA) [BAF+14] to support spatial data. IDAA is a hybrid system, which uses DB2® for z/OS® [IBM14b] for transactional workload. A copy of the data resides in the accelerator, which is based on Netezza technology [Fra11], and deals with analytical workload in a

*IBM Germany Research & Development, Böblingen, Germany

†IBM Silicon Valley Lab, San Jose, USA

high performing way. The DB2 optimizer is responsible for the query routing decision, i. e. whether to run the query in DB2 itself or to offload it to the accelerator. For us, this architecture is a stepping stone towards a single operational data store [Inm99]. The benefits can be found in an overall reduced complexity of the IT infrastructure and better integration, all reducing costs for exploiters.

Many application areas take advantage of storing, retrieval, and querying of geometric, geographic, and spatial data.[1] The data space for geographic data is the surface of the Earth, possibly as a two-dimensional abstraction (projection). In geographic contexts, it models static objects in the real world like streets, buildings, political boundaries, etc. Moving objects like ships or cars with a changing location can also be described. Other examples are disaster control and emergency management requiring information about the exact location of a catastrophe or accident and details about the available infrastructure like roads and railways. Yet another important user group are military entities, who use spatial data for centuries already for their strategic and tactical planning.



Figure 1: Visualized Map Data Extracted from a DB2 Database System

DB2 for z/OS as well as Netezza already have support for spatial data. Both follow the SQL/MM: Spatial standard [ISO03a, Sto03]. However, there are still differences in the scope of supported functionality and the storage of spatial objects, especially size limitations. Those differences must be solved together with processing of spatial queries in IDAA and high performing data ingestion for a fully functional product feature.

In all our efforts, the overriding design principle for providing acceleration capabilities of spatial data was to avoid changes or requiring rewrites of an application's SQL statements. A spatial query send to DB2 is rewritten by the DB2 optimizer to the Netezza SQL dialect and then passed on to IDAA, which delegates the actual execution to the Netezza backend. The data structures in Netezza are designed in a way that is just based on function mapping, but without any further changes to the original SQL statement. That means, DB2 is not responsible for introducing additional joins, for example.

The remainder of the paper is structured as follows. The architecture of the IBM DB2 Analytics Accelerator is summarized in section 2. Section 3 gives a very brief overview on Netezza spatial and DB2 spatial, highlighting details that are important for the integration with IDAA. Our new approach to deal with space limitations in Netezza and the related integration of spatial support in IDAA are presented in section 4, together with

[1] Henceforth, the terms geometry, spatial data, and spatial information are used synonymously.

some preliminary performance measurements based on a prototype implementation currently under way. Finally, the paper concludes in section 5 with a summary and general outlook to future direction for the development of this new product feature.

2 Overview on the IBM DB2 Analytics Accelerator

IDAA [BAF+14] is based on the Netezza appliance [Fra11], which is used as backend. It provides the data storage capabilities to manage large amounts of data and to provide exceptional performance for querying that data. IDAA's functionality is extended over time to better satisfy customer requirements. [Sto13]

Figure 2 illustrates the high-level architecture of IDAA, also called *Data Warehouse Accelerator*. An additional process (*DWA*) – that implements the integration layer with DB2 for z/OS – runs on the Netezza hardware and operating system. This integration layer is the entry point for all requests originating either from DB2 or IDAA stored procedures running in DB2. The DRDA protocol [DRD03] is used for the communication between both hardware platforms. Requests to execute queries are passed from DB2 to the Netezza backend by translating DRDA protocol to CLI/ODBC. Administrative requests, e. g., to provide a list of accelerated tables, are handled in DWA itself. If necessary, SQL queries against the Netezza backend are executed to collect backend-related meta data and/or statistical information.

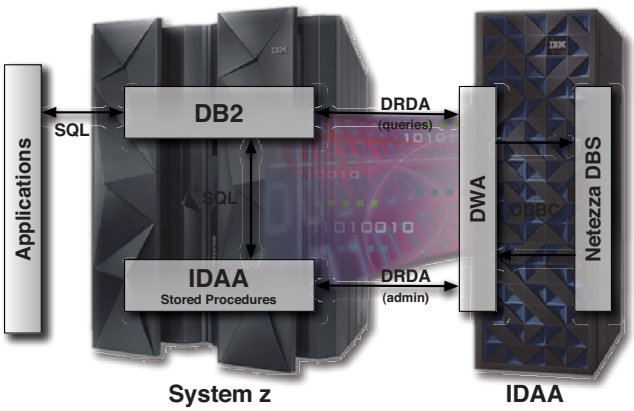


Figure 2: IDAA System Architecture

It is possible to associate multiple accelerators with a single DB2 system in order to establish an environment that supports high availability and disaster recovery. An appropriate workload balancing is applied by DB2 in case the connected accelerators have different hardware and workload characteristics. Similarly, a single accelerator can be connected to multiple DB2 systems, sharing its resources. Another workload balancing layer is applied on that level.

Aside from query processing, it is also necessary to maintain and refresh the snapshot of the data in the accelerator in case the original data in DB2 changes. IDAA offers 3 options for the data refresh, which are explained in more detail.

The IBM DB2 Analytics Accelerator comes with an extremely fast loader for whole tables or (a set of) partitions thereof. Refreshing the data of an entire table (cf.figure 3) is typically done for rather static data and for non-partitioned tables. Partition-based load (cf.figure 4) is targeted at partitioned tables, where updates are performed rather infrequently and only to a small subset of partitions.

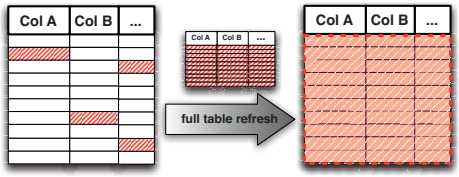


Figure 3: Full Table Refresh

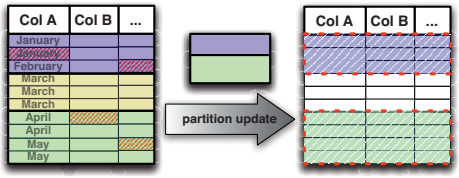


Figure 4: Partition Update

If a low latency for the data currency in IDAA is not very important and queries return acceptable results, even if the data is slightly outdated by a few minutes or hours, both options are viable. Additionally, the initial load of the data into the accelerator is accomplished that way with very good performance, exceeding a throughput of 1.5 TB/h and maxing out at the network bandwith between DB2 and IDAA.

For tables with a higher update frequency and where a high data currency on the accelerator is desired, a third option for data maintenance is the Incremental Update feature (cf. figure 5). Incremental update is based on replication technology [BNP+12], which reads DB2 logs and extracts all changes to accelerated DB2 tables. A latency of about 1 minute is achieved, which is usually fast enough for reporting systems and data warehouses. That is even more true considering complex accelerated queries may take several minutes (or hours) in DB2 and still a few seconds with IDAA.

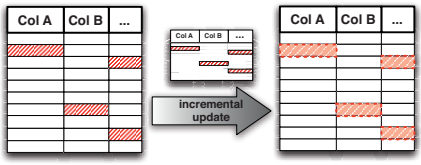


Figure 5: Refreshing Table Data With Incremental Update

It is the responsibility of the database user/administrator to trigger the refresh of the data in each accelerator individually (or to set up incremental update where appropriate). Since many customers have thousands of tables in DB2 and also in IDAA, it is important to understand query access patterns to the individual tables and monitoring the accelerated workload.

3 Spatial Data

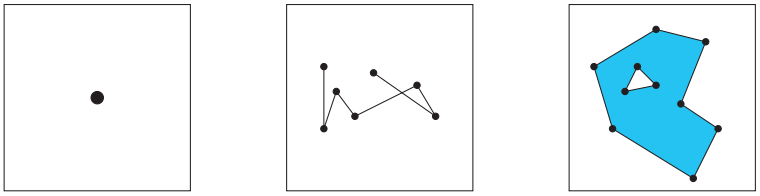


Figure 6: Spatial Primitives

Points, linestrings, and polygons are the basic building blocks for storing spatial data in relational database systems. The geometries are treated as scalar values with a dedicated set of functions. Some systems, e. g. DB2 for LUW, employ structured types and establish a hierarchy while others just have a single data type. Other systems use large objects (BLOBs) and encode all details required to represent the geometry in an opaque way. This does not require complex infrastructure for object-relational extensions. DB2 for z/OS and Netezza have gone that route. [Sto06]

There are over 100 spatial functions, which can loosely be categorized in:

- convert between geometries and external data formats,
- retrieve properties/attributes or measures from a geometry,
- compare two geometries with respect to their spatial relationship, and
- generate new geometries from others.

Probably, the most important category is the comparison based on a spatial relationship since those are used for *spatial joins*. Specific functions exist to test whether 2 geometries are disjoint, touch each other, overlap, or if one is completely contained in the other.

A few systems have extended their spatial support even further and introduced topology and network models [Ora05]. Those features will not be further discussed here.

3.1 Spatial Support in DB2 z/OS vs. Netezza

DB2 for z/OS has introduced a built-in data type `ST_Geometry` [IBM14a], which is internally a derivate of a BLOB. The binary representation holds all information about the specific type (point vs. linestring vs. polygon), the actual coordinates of the points defining the geometry, and additional attributes like the minimum and maximum X and Y coordinates. The (compressed) binary representation of geometries is limited in size to a maximum of 2 MB. This is sufficient for most practical considerations.

Netezza does not support large objects or structured types. Instead, Netezza's spatial feature [IBM14c] stores each geometry as a scalar value using `VARBINARY` underneath. The

maximum size for a scalar value is limited by the 64K page size. The size and precision of geometry data can vary considerably. There are geometries that exceed the page size limit very quickly, e. g. one instance of the border lines of the USA needs 300 K (compressed) alone. In such cases, the too large geometries are split (chopped) into multiple, smaller but consistent geometries when the data is inserted into the Netezza tables by a separate tool [Sof14]. Naturally, that splitting adds overhead.

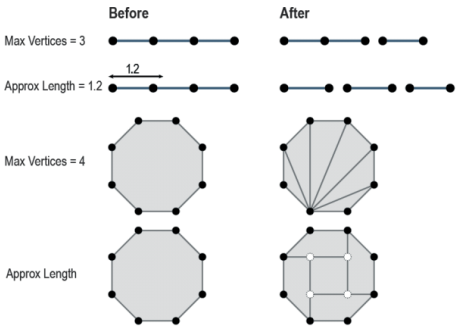
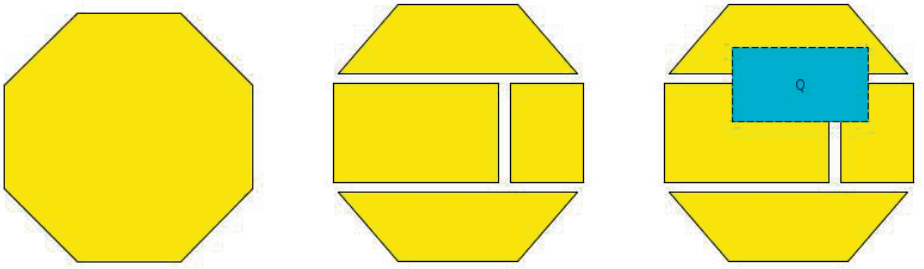


Figure 7: Example for Chopping Geometries (Source: [Sof14])

Furthermore, not all spatial functions can be applied correctly to the individual, smaller geometries. An example is shown in figure 8. Assuming that the octogon in figure 8(a) is actually a much more complex geometry, it is broken down into several smaller geometries as figure 8(b) demonstrates. Attempting to query those multiple geometries may yield incorrect results as is illustrated using figure 8(c). If the spatial predicate is `ST_Within` and tests whether the blue query rectangle is *fully* contained within the (yellow) geometry, the answers on the smaller geometries are all `false` and there is no way to determine the correct answer of `true` without recomposing the original geometry. Due to these issues, Netezza does not provide all spatial functions.



(a) Original Geometry (b) Multiple Smaller Geometries (c) Sample Query

Figure 8: Functional Problems with Breaking Down Geometries

Our approach below avoids the functional problems for spatial support in IDAA. We take advantage of user-defined aggregate functions, table functions, and the presence of IDAA as a layer on top of the Netezza database system, which is running on the same system as Netezza itself. That allows us to properly handle geometries of all sizes when loading data from DB2 into IDAA and also when querying the data with any of the spatial predicates supported by DB2.

4 Integrating Spatial Data in IDAA

The fundamental differences for spatial support in DB2 and Netezza (cf. section 3.1) are really major obstacles. Breaking down and splitting geometries looses key functionality. Additionally, the overhead for the splitting during ingestion is not acceptable from a performance point of view. Therefore, our chosen path is not a light-weight integration of the existing products but rather a deep integration of spatial functionality in IDAA.

4.1 Storage in Netezza

Our first step resolves the size limitations of Netezza spatial. Naturally, it is still necessary to break down large geometries in such a way that each part can fit into a single scalar `VARBINARY` value. But instead of constructing several smaller, consistent geometries, we take the *binary* representation of the geometry in DB2 and chunk that binary representation. Figure 9 illustrates this. The individual parts (or chunks) are all equally sized (except the last one), and each chunk fits into a VARBINARY value.

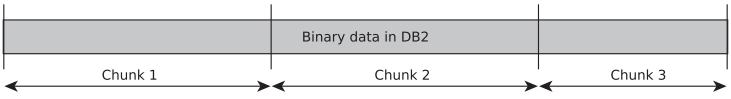


Figure 9: Binary Chunking for Netezza

The first consequence of the chunking is that we can usually not store a geometry together with all its non-spatial attributes (e. g. street name) in one relational table. The chunked geometries are stored in an additional *side table*.[2] Denormalization would be an alternate, but not desirable option.

```
-- user table
CREATE TABLE country (
     id INTEGER NOT NULL PRIMARY KEY,
```

[2]Side tables are also the means used by Netezza spatial for the same purpose.

```
    name VARCHAR(100) NOT NULL,
    abbreviation VARCHAR(3) NOT NULL,
    -- other attributes
);

-- side table
CREATE TABLE country_side (
    country_id INTEGER NOT NULL
       FOREIGN KEY REFERENCES country,
    chunk VARBINARY(64000) );
```

Listing 1: Sample Schema Definition for side tables

Theoretically, it would be possible to store (large) geometries only in the file system and a reference in the user table [ISO03b]. The side table could be avoided. However, all issues related to managing data external to the database system (like proper cleanup, replication for high availability, security, ...) come into play. The complexity and implementation efforts exceed the benefits by a fair margin.

4.2 Spatial Data Ingestion

With the storage structure being defined, the first question arises with the insertion of the data into the Netezza tables when copying it from DB2. While the chunking could be done by IDAA itself, the same functionality will be needed in other contexts as well. For instance, if the spatial function `ST_Union` is used in a query, the resulting geometry may become too large and exceed the 64K page size limit again. If that geometry shall be inserted into another Netezza table, it has to be chunked. Therefore, we have opted to implement the chunking via SQL table functions in Netezza itself. The table function `ChunkGeometry` receives the geometry as input and simply generates the run of contiguous chunks where each chunk has an appropriate length. Those chunks are inserted into the side table

```
INSERT INTO country_side SELECT * FROM
TABLE ( ChunkGeometry(id, <geom>) ) AS chunks(id, chunk)
```

Listing 2: Inserting geometries in side tables

Geometry Locators Since the chunking is realized by IDAA in the Netezza backend itself, it is necessary to transfer the geometry data to Netezza before the table function can be called. But Netezza does not allow for any data value to exceed the page size limit. In order to still be able to transfer larger values from IDAA to Netezza, we use *geometry locators* – a technique similar to LOB locators.

A geometry locator (cf. figure 10) is a means to describe different storage formats. The first byte in the locator's data identifies the specific format. If a geometry is small enough,

it is directly embedded (*inline*) and we use an indicator of 0x00. Larger geometries are stored in a file in the file system (or in memory). The file based transfer informs the `ChunkGeometry` function about the file name and the function reads the data from there based on indicator 0x01.

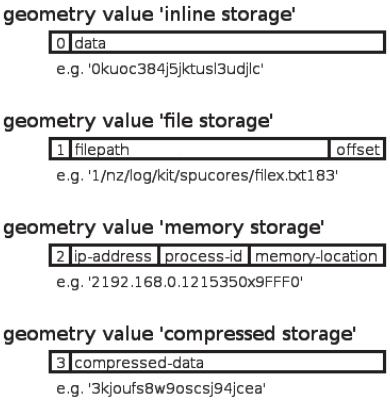


Figure 10: Geometry Locator

Of course, other options are possible and figure 10 also shows a compressed inline storage. Although, the geometry data is already compressed, it could happen that the binary data exceeds the page size limit only slightly. Another compression step, e. g. using LZW [Wel84], may reduce the data size just by those few bytes that are needed to squeeze in the geometry – and, thus, avoid indirections as with the other storage formats.

Note that the (simple) file system based approach works because the IDAA server process runs on the same system as the Netezza backend. Thus, both components have access to the same file system. If that were not the case, a shared file system or remote file transfer could be used.

Evaluation As of today, we have implemented a prototype of the `ChunkGeometry` function and use it to insert the spatial data from IDAA. The binary chunking beats FME's [Sof14] chopping into small, consistent geometries. FME needed more than 2 hours for 23K geometries (on a very slow system), while our approach finished in 20 minutes. But note that we have not conducted any serious performance comparison yet.

4.3 Query and Result Set Processing in IDAA Context

The binary chunking comes with a penalty during query processing: whenever predicates shall be evaluated on a geometry or other spatial functions are used, it is mandatory to recompose the original geometry. This is accomplished using our new aggregate function

RecomposeGeometry. The aggregate gets all chunks of the geometry as input (as individual rows from the side table) and internally constructs the full geometry. The result is a geometry locator.

Recomposition is only needed as the first step in query processing. Once a chunked geometry is recomposed and its geometry locator exists, all stacked spatial functions work with that locator. Also, the locator is used when binding out the geometry from Netezza to IDAA.

IDAA receives its queries from DB2, and DB2 is not aware of the storage details for spatial data in the Netezza backend. But we want to avoid complex logic for query rewrite in DB2 itself. For each accelerated table with spatial data, we have the above mentioned side table. We use a view to hide this side table and to inject the RecomposeGeometry aggregate at the same time. The view takes the place of the actual accelerated table, so any query sent by DB2 to IDAA is not even aware that a view is accessed. Netezza resolves the view transparently into the join over both tables.

```
CREATE VIEW country_table(id, name, abbreviation, geom) AS
SELECT c.id, c.name, c.abbreviation,
     RecomposeGeometry(s.chunk)
FROM country AS c JOIN country_side AS s
       ON ( c.id = s.country_id )
GROUP BY c.id;
```

Listing 3: View to hide side tables and recompose aggregate function

For example, a SQL query in DB2 like SELECT ST_Buffer(geom, 5, 'Miles') FROM country WHERE ... needs to be rewritten to access the specific table name used in Netezza (country_table in our example), and that's nearly all there is to it. Thus, we have achieved a very simple decoupling of DB2 from the IDAA and Netezza internal storage mechanism for the spatial data.

We have mentioned above that geometry values are bound out from Netezza to IDAA using geometry locators. Thus, the result set of the query contains a locator. Given that result sets are always consumed by IDAA, IDAA has to resolve the locator and extract the real geometry either from it directly (if inline) or by accessing the shared file system. This is the reverse procedure as for spatial data ingestion into the Netezza backend.

Evaluation Our prototypical implementation has reached the point where we can verify the functionality. So far, it has proven that we can implement all spatial functions available in DB2 with the correct semantics. The next steps will be a performance evaluation to measure the impact and acceleration for query processing. Due to the additional join with the side table, we do not expect to achieve the same very good acceleration factors of 10x-100x as IDAA delivers for non-spatial queries. However, our expectations are still such that spatial queries scanning over TBs of data should still run very well in our environment.

5 Summary and Outlook

In this paper we have presented a prototype implementation for the integration of spatial data support into IDAA. The main issue are limitations of the size for `VARBINARY` values in Netezza, which we resolve by chunking the binary representation of large geometries and storing those chunks in a side table, which is always joined at query time. We have proven that functional limitations of Netezza spatial can be resolved this way.

Our next steps in this project are more in-depth performance evaluations. We want to experiment with additional storage formats to exchange large geometries between IDAA and Netezza as well as passing such geometries from one spatial function to the next within the same SQL statement. Although, we always keep the geometry values stored in the Netezza tables, we will evaluate whether caching of recomposed geometries can be done in order to speed up the recomposition steps. The primary question with such a (file system based) cache will become the maintenance, i. e. which geometry objects to cache and which to discard in order to stay within the available resources like disk space.

6 Trademarks

IBM, DB2, and z/OS are trademarks of International Business Machines Corporation in USA and/or other countries. Other company, product or service names may be trademarks, or service marks of others. All trademarks are copyright of their respective owners.

References

[BAF+14] P. Bruni, J. Arnold, W. Favero, L. Cruz, J. Feinsmith, A. Griner, J. Guo, C. Harlander, J. Kern, R. Kumar, R. Li, A. Perkins, J. Sloan, S. Speller, and D. Tonelli. *Reliability and Performance with IBM DB2 Analytics Accelerator V4.1*. IBM Redbooks, 2014.

[BNP+12] A. Beaton, A. Noor, J. Parkes, B. Shubin, C. Ballard, M. Ketchie, F. Ketelaars, D. Rangarao, and W.V. Tichelen. *Smarter Business: Dynamic Information with IBM InfoSphere Data Replication CDC*. IBM Redbooks, 2012.

[DRD03] The Open Group. *DRDA V5 Vol. 1: Distributed Relational Database Architecture*, 2003.

[Fra11] P. Francisco. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*. IBM Redbooks, 2011.

[IBM12] IBM. IBM Banking and Financial Markets Data Warehouse V8.5. Technical report, 2012. http://www.ibm.com/software/data/industry-models/financial-markets/.

[IBM14a] IBM. *Administering IBM Spatial Support for DB2 for z/OS*, 2014.

[IBM14b] IBM. *DB2 11 for z/OS*, 2014. http://www.ibm.com/support/knowledgecenter/api/content/SSEPEK/db2z_prodhome.html.

[IBM14c] IBM. *IBM Netezza Analytics – In-Database Analytics Developer's Guide, Release 3.0.1*, 2014.

[Inm99] W. Inmon. *Building the Operational Data Store*. John Wiley & Sons, 1999.

[ISO03a] ISO/IEC 13249-3:2003. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 3: Spatial*, 2nd edition, 2003.

[ISO03b] ISO/IEC 9075-9:2003. *Information Technology – Database Languages – SQL – Part 9: Management of External Data (SQL/ MED)*, 2nd edition, 2003.

[Leh03] W. Lehner. *Datenbanktechnologie für Data-Warehouse-Systeme – Konzepte und Methoden*. Dpunkt Verlag, 2003.

[Ora05] Oracle Corp., Redwood City, CA, USA. *Oracle Spatial – User's Guide and Reference, 10g Release 2 (10.2)*, 2005.

[Sof14] Safe Software. *FME Workbench Transformers*, 2014. http://docs.safe.com/fme/html/FME_Transformers/Default.htm.

[Sto03] K. Stolze. SQL/MM Spatial – The Standard to Manage Spatial Data in Relational Database Systems. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz*, Lecture Notes in Informatics, Leipzig, Germany, 2003. Springer-Verlag.

[Sto06] K. Stolze. *Integration of Spatial Vector Data in Enterprise Relational Database Environments*. PhD thesis, Database and Information Systems Group, University of Jena, Germany, 2006.

[Sto13] K. Stolze. Seamless Integration of Archiving Functionality in OLTP/OLAP Database Systems Using Accelerator Technologies. In *BTW 2013, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 15. BTW-Konferenz*, Lecture Notes in Informatics, Magdeburg, Germany, 2013. Springer-Verlag.

[Wel84] T. A. Welch. A Technique for High-Performance Data Compression. *j-COMPUTER*, 17(6):8–19, 1984.

Systems and Applications

Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue?

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Thomas Hauf
Hochschule Darmstadt
thomas.hauf@stud.h-da.de

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Abstract: NoSQL data stores are becoming increasingly popular in application development. These systems are attractive for developers due to their ability to handle large volumes of data, as well as data with a high degree of structural variety. Typically, NoSQL data stores are accessed programmatically. Due to the imminent lack of standardized query languages, building applications against the native interfaces of NoSQL data stores creates an unfortunate technical lock-in. To re-gain platform independence, developers turn to object mapper libraries as an additional level of abstraction when accessing NoSQL data stores.

The current market for Java object mappers that support NoSQL data stores is still volatile, with commercial and open source products competing for adoption. In this paper, we give an overview on the state-of-the-art in Object-Relational Mappers that can handle also NoSQL data stores, as well as dedicated Object-NoSQL Mappers.

We are able to show that choosing the right object mapper library is a strategic decision with far reaching consequences: Current mappers diverge in the NoSQL data stores that they support, in their features, their robustness, their truthfulness to the documentation and query standards, and ultimately, in the runtime overhead that they introduce. Especially in web development, runtime overhead is a crucial aspect contributing to the application latency, and ultimately, the user experience. By shedding light on the current market, we intend to provide software architects with the necessary information to make informed decisions.

1 Introduction

During the last decade, we have seen radical changes in the way software is being built: Where traditional shrink-wrapped software undergoes yearly releases, new versions of cloud-based applications are released on a weekly if not daily basis (quoting Marissa Mayer in [Lig10]). This goes hand in hand with developers striving to be agile. In the spirit of lean development, design decisions are made as late as possible, a strategy that also applies to the design of the schema. Actually, the schema-flexibility of many NoSQL data stores is a driving force behind their popularity, even for applications where the expected data volume by itself does not justify using a NoSQL data store.

However, NoSQL data stores bring about their own challenges: As of today, there is no

```
1  @Entity
2  public class Profile {
3      @Id
4      int profileID;
5
6      String firstname;
7      String lastname;
8      int year;
9      String country;
10     /* not showing the methods */
11 }
```

Figure 1: A Java class declaration of user profiles with JPA annotations.

standardized query interface. When building applications against NoSQL systems, calling the proprietary APIs condones a technical lock in. Especially in a new and volatile market such as today's NoSQL data stores, this can put the long-term success of a project at risk.

To avoid dependency on a particular system or vendor, developers commonly rely on object mapper libraries. These mappers build on the tradition of Object-Relational Mappers (ORMs) designed for interfacing with relational databases. They handle the mundane marshalling between objects in the application space and objects persisted in the data store. For instance, Figure 1 shows a Java class declaration for user profiles. The object mapper annotation `@Entity` conveniently declares that instances of this class can be persisted.

When building applications against relational databases, software architects can choose from a range of established and well-documented ORMs [MW12]. Traditionally, these ORMs handle the impedance mismatch between object-oriented programming languages and the relational model. In programming with Java, the Java Persistence API (JPA) including the Java Persistence Query Language (JPQL) has become state-of-the-art. JPA and JPQL are "standardized" within the Java Community Process [Jav09]. Using these APIs, database applications can become largely independent of ORM vendors.

It is a natural consequence that with the rising popularity of NoSQL data stores, some ORM vendors are extending their support to NoSQL data stores. Yet there are also new players in this market, offering special-purpose Object-NoSQL Mappers.

To the software architect, this raises beguiling questions:

- Which features are desirable in a mapper library when building applications against NoSQL data stores?
- Are the products offered today mature enough to be employed in *professional* application development?
- What is the runtime overhead imposed by object mapper libraries?

To answer these questions, we study a representative sample of Object-NoSQL Mappers (ONMs) for Java development as the main contribution of this article.

Structure. After reviewing the basics of NoSQL data stores and Object-NoSQL mapping in Section 2, we give an overview of the state-of-the-art and provide a comparative study of Object-NoSQL Mappers in Section 3. Besides the generic mapping capabilities, we analyze in Section 4 whether any schema management tasks are supported by Object-NoSQL Mappers beyond what the underlying NoSQL data store supports. The results of our performance evaluation are presented in Section 5. We then conclude with a summary and an outlook on future developments.

2 Foundations

We briefly recount the variety of NoSQL data stores, and proceed with an overview over Object-NoSQL mapping.

2.1 NoSQL Data Stores

NoSQL data stores vary in their data and query model, scalability, architecture, and persistence design. The most common categorization is by data model, distinguishing *key-value stores*, *document stores*, *column-family stores*, and *graph databases* [EFH+11, Tiw13].

Key-value stores persist pairs of a unique key and an opaque value.

Document stores also store key-value pairs, yet the values are structured as "documents". This term connotes loosely structured sets of name-value pairs, usually in JSON (JavaScript Object Notation) [Ecm13] format or the binary representation BSON, a more type-rich format. Name-value pairs represent the properties of data objects. Names are unique within a document. Since documents are hierarchical, values may not only be scalar or appear as lists, but may contain nested documents.

Column-family stores manage records with properties. A schema declares property families, and new properties can be added to a property family ad hoc, on a per-record basis.

Graph databases provide operations on a graph data model with nodes (representing entities) and edges (representing relationships).

Object-NoSQL Mappers are available for all types of NoSQL data stores. In Section 3 we analyze selected object mapper libraries.

2.2 Object-NoSQL Mapping

From the application developer's point of view, Object-NoSQL Mappers follow the same ideas as Object-Relational Mappers, typically relying on annotations within class declarations in the application source code. We illustrate these annotation principles using a small social network example which will be extended in the next sections. Let us consider

Profile

profileID	firstname	lastname	year	country
4711	Miroslav	Klose	1978	DE

(a) A persisted object in a column-family store.

```
{
  "profileID" : 4711,
  "firstname" : "Miroslav",
  "lastname"  : "Klose",
  "year"      : 1978,
  "country"   : "DE"
}
```

(b) A persisted object in a document store.

Figure 2: The class declaration from Figure 1 implies the schema of persisted objects.

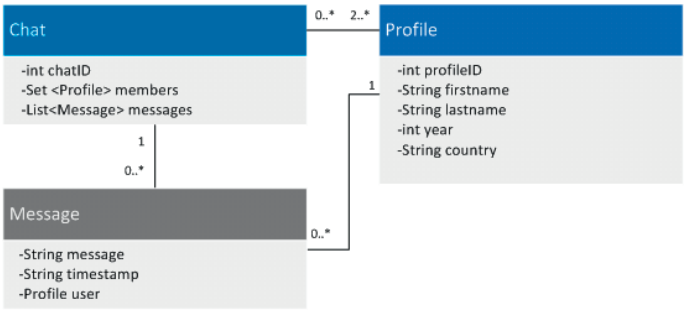


Figure 3: Class diagram for example scenario (extract).

Figure 1, illustrating the use of annotations in Java Persistence API (JPA) [Jav09]. Class `Profile` captures a user profile with a unique ID and user information (e.g., firstname and lastname). Due to annotation `@Entity` in line 1, `Profile` objects can be persisted. Annotation `@Id` in line 3 declares a class member attribute as the identifying key.

Figure 2 shows how instances of this class are persisted in different types of NoSQL data stores. Subfigure (a) shows an object persisted in a column-family store and subfigure (b) the same object persisted as JSON document in a document store. Evidently, the class declaration with the object mapper annotations also declares the schema of persisted objects.

Likewise, associations between classes can be annotated as *relationships* and carry cardinalities: Typically one-to-one, one-to-many, and many-to-many relationships are supported. To illustrate this, we extend our example by classes `Chat` and `Message`. Users can communicate together via messages within chats. Figure 3 shows the class diagram. `Message` objects have the same lifetime as the associated `Chat` object. We will come back to this point later in the discussion. Figure 4 contains the corresponding Java class declarations with JPA annotations. Line 6 of Figure 4(a) declares the `@ManyToMany` annotation for the many-to-many relationship between `Chat` and `Profile`. Analogously, the `@ManyToOne` association between `Message` and `Profile` is shown line 8 of Figure 4(b). Relationships can be unidirectional or bidirectional. Because we implement these relationships as unidirectional, no changes to class `Profile` are required (Figure 1).

```
1  @Entity
2  public class Chat {
3      @Id
4      int chatID;
5
6      @ManyToMany
7      Set<Profile> members;
8
9      @ElementCollection
10     List<Message> messages;
11 }
```

(a) Class declaration with `@ManyToMany` and `@ElementCollection` annotations.

```
@Embeddable
public class Message {


    String message;

    String timestamp;

    @ManyToOne
    Profile user;
}
```

(b) Embeddable class declaration with `@ManyToOne` annotation.

Figure 4: Java class declarations with different JPA annotations for relationships.

Another interesting point is the implementation of the one-to-many relationship between `Chat` and `Message`. Instead of using the `@One-To-Many` annotation, we use an ElementCollection. An ElementCollection in JPA defines a collection of instances of a basic type or embeddable class. The entities of the embeddable class have the same lifetime as their owner. Furthermore it is not necessary to manage a separate id for the instances of the embeddable class. An ElementCollection is declared via the `@ElementCollection` annotation in the owner class (see line 9 in Figure 4(a)) and the declaration `@Embeddable` for the embeddable class (see line 1 in Figure 4(b)).

Today, there is a large body of experience how to map from the object-oriented model to the relational database model, and most Object-Relational Mappers implement similar mapping strategies. In contrast, the mapping from an object-oriented class model to a NoSQL data store depends strongly on the underlying NoSQL data model. Furthermore, different mappings are possible within the same NoSQL data model (c.f. Section 2.1). For example, collections as well as one-to-many relationships can be embedded or referenced. An exhaustive discussion of mapping variants is therefore beyond the scope of this paper, and remains an understudied research area as of today. We point out interesting current limitations in mapping of some object-oriented modeling constructs in Section 3.

3 Comparison of Object-NoSQL Mappers

We now state our desiderata for Object-NoSQL Mappers (ONMs). After that, we present a selection of state-of-the-art ONM products and evaluate them w.r.t. our desiderata.

3.1 Requirements

Naturally, Object-NoSQL Mappers provide basic CRUD (Create, Read, Update, Delete) operations. In the interest of platform and vendor independence, the ONM should further offer a standardized query language. Yet for performance reasons, in particular to leverage proprietary APIs, it can be worthwhile if the ONM allows access to the query language native to the NoSQL data store.

If values are modified or objects are destroyed within the application, object mapper libraries generate the corresponding data store update respectively delete statements. Hence, updates and delete operations are usually executed on a single object. But for performance reasons it should be possible to execute update and delete operations on several objects in batch. Therefore, ONMs should support appropriated update and delete statements. Also for performance reasons, batch or bulk inserts should be provided.

NoSQL data stores can handle large volumes of data, exploiting massive parallelism, typically using MapReduce [DG04]. Therefore, ONMs should support MapReduce or some other parallel programming approach. Finally, in NoSQL application scenarios the usage of several data stores for different data within the same application become an important request. This scenario is described by the term *polyglot persistence*, coined by Martin Fowler [SF12]. Ideally, ONMs enable polyglot persistence, i.e. the usage of different NoSQL or relational data stores within the same application.

3.2 Overview over Object-NoSQL Mappers

The market for Object-NoSQL Mappers is still volatile. Various libraries are available, among them open source projects which no longer seem to be maintained. We therefore restrict our evaluation to projects where the last stable release is not older than one year.

For the programming language Java, the standardized Java Persistence API (JPA) with the Java Persistence Query Language (JPQL) has become state-of-the-art [Jav09]. Due to this fact and the popularity of Java in application development, we focus on Java ONMs.

We distinguish ONMs which support several NoSQL data stores (*Multi Data Store Mapper*) and ONMs which support only a single system (*Single Data Store Mapper*). While Single Data Store Mappers may cause a technological lock in, they offer interesting features and may display superior runtime performance.

With *EclipseLink* [Ecl14] and *DataNucleus* [Dat14a], we consider two very prominent libraries for object-relational and object-XML mapping that have been extended for object-NoSQL mapping. Hibernate, in particular, is one of the most popular ORMs. However, in contrast to EclipseLink and DataNucleus, object-NoSQL mapping has not been integrated into its main object mapper library yet. Instead, *Hibernate OGM* [Red14a] comes as its own library. *Kundera* [Imp14b] by Impetus is a ONM without "ORM-history" and was one of the earliest adopters in this market. *Morphia* [Mon14] is a proprietary Single Data Store Mapper for MongoDB. All mentioned mappers are available as open source.

	Multi Data Store Mapper				
	Hibernate OGM	Kundera	DataNucleus	EclipseLink	Morphia
Evaluated version	4.1 B 6	2.13	4.01	2.5.2	0.108
Key-value stores					
Infinispan	✓	–	–	–	–
Ehcache	✓	–	–	–	–
Redis	–	✓	–	–	–
Elasticsearch	–	✓	–	–	–
Oracle NoSQL	–	✓	–	✓	–
Document stores					
MongoDB	✓	✓	✓	✓	✓
CouchDB	✓	✓	–	–	–
Column-family stores					
Cassandra	–	✓	✓	–	–
HBase	–	✓	✓	–	–
Graph databases					
Neo4j Embedded	✓	✓	✓	–	–

Table 1: Distinguishing ONMs by the supported NoSQL data stores.

3.3 Feature Analysis

Table 1 lists the evaluated Object-NoSQL Mappers with the supported NoSQL data stores. Notably, the popular MongoDB system is supported by all evaluated ONMs.

Table 2 compares the ONMs by our desiderata. Basic **CRUD** operations are more or less supported by all evaluated products.

However, there are some modeling restrictions regarding *ElementCollections* (c.f. Section 2): DataNucleus does not support ElementCollections for CouchDB, Cassandra, HBase, and Neo4j [Dat14a]. Kundera does not support ElementCollections for CouchDB or Neo4j, yet this limitation is not evident in the current Kundera documentation. To handle this drawback, the elements of the collection have to be modeled as non-embedded entity class with own identity and a one-to-many relationship between the associated classes. Figure 5 shows the modified modeling of relationship between `Chat` and `Message` classes (c.f. Figure 4). Line 1 of Figure 5(b) contains the `@Entity` annotation instead of `@Embeddable` and line 3 and 4 the now necessary identifying attribute with its JPA annotation. The relationship between `Chat` and `Message` is now annotated with `@OneToMany` instead of `@ElementCollection` (line 9 in Figure 5(a)). As a consequence the application developer is now responsible to implement the lifetime dependencies between messages and chats in the application.

	Multi Data Store Mapper				
	Hibernate OGM	Kundera	DataNucleus	EclipseLink	Morphia
Create					
Single object	✓	✓	✓	✓	✓
Batch insert	✓	✓	✓	✓	✓
Read					
Query Language	JPQL*	JPQL*	JPQL, JDOQL	JPQL*	proprietary
Native Queries	✓	✓	–	✓	✓
MapReduce	–	–	–	–	✓
Update					
Single object	✓	✓	✓	✓	✓
Multiple objects	–	✓	–	✓	✓
Delete					
Single object	✓	✓	✓	✓	✓
Multiple objects	–	✓	✓	✓	✓
Polyglot Persistence	–	✓	(✓)	✓	–

Table 2: CRUD operations supported by Object-NoSQL Mappers.

Query Language Support Object-Relational Mappers target relational data stores, where we can rely on SQL as a standardized and well-understood query language. While not all relational databases implement the full SQL standard, and many vendors add their own extensions, there is nevertheless conformity in supporting a large, common SQL fragment represented within a object query language like JPQL. In contrast to relational databases, there is no standardized access to NoSQL data stores. Systems vary greatly, even in how they implement CRUD operations. Most systems do not support any joins. Many do not offer aggregate functions or the `LIKE` operator, again, some systems do.

This raises the question how Multi Data Store Mappers deal with this heterogeneity. Overall, there seem to be three approaches,

1. to offer only the particular subset of features that is implemented by *all* supported NoSQL data stores, i.e. the intersection of features, or
2. to distinguish by data store and to offer only the set of features implemented by a particular NoSQL data store, or
3. to offer the same set of features for all supported NoSQL data stores, possibly complementing missing features by implementing them inside the ONM library.

In the first approach, the ONM offers only the query language constructs translatable to query operators implemented by all supported NoSQL data stores. While we can write data store independent applications, the query capabilities are severely limited.

The second approach is to offer query language operators individually, depending on the functionality of the underlying NoSQL data store. This approach is chosen by Hibernate OGM, Kundera, and EclipseLink (marked with a * in Table 2). We can now make use of the full set of features implemented in each data store. However, this puts portability at

```
@Entity
public class Chat {
    @Id
    int chatID;

    @ManyToMany
    Set<Profile> members;

    @OneToMany
    List<Message> messages;
}
```

(a) Class declaration with `@OneToMany` annotation.

```
@Entity
public class Message {
    @Id
    int messageID;

    String message;
    String timestamp;

    @ManyToOne
    Profile user;
}
```

(b) Class declaration with `@Entity` instead of `@ElementCollection`.

Figure 5: Alternative modeling of `Chat-Message` relationship.

	Cassandra	HBase	MongoDB	CouchDB	Redis	Oracle-NoSQL	Neo4j
ORDER BY	–	–	✓	–	–	–	–
AND	✓	✓	✓	✓	✓	✓	✓
OR	– (✓with Lucene)	✓	✓	–	✓	✓	✓
BETWEEN	✓	✓	✓	✓	✓	✓	✓
LIKE	– (✓with Lucene)	✓	✓	–	–	–	✓
IN	–	–	–	–	–	–	–

Table 3: Kundera's support for JPQL constructs is data store specific [Imp14a].

risk: A feature supported in one data store may be not supported in another data store. For instance, in Kundera the `ORDER BY` operator is supported for MongoDB only [Imp14a]. Let us go into more detail for Kundera. As seen in Table 3, Kundera does not support the same set of JPQL constructs for all data stores. Hence, with this approach, application developers have to choose between functionality and portability.

To overcome these limitations, some ONMs compromise between the second and the third approach. They use third-party libraries to offer more functionality for some but not for all supported NoSQL data stores. As we can also see in Table 3, Kundera provides the `LIKE` operator for Cassandra although this operator is not natively supported. To do so, Kundera relies on Apache Lucene for an implementation of the `LIKE` operator [Imp14a].

Similarly, Hibernate OGM offers Hibernate Search [Red14c] to index and query objects (entities), as well as to run full-text queries. The latter feature is driven by Apache Lucene, extended with an object oriented abstraction including an object oriented, domain-specific query language [Red14a]

The third and most advanced approach is to offer the same set of query operators for *all* NoSQL data stores. If a feature is not supported by a data store it has to be realized within the Object-NoSQL Mapper. This allows for portable applications using the full power of the standardized query language offered by the ONM. Naturally, when an operation is implemented outside of the data store, runtime performance may decrease.

In this spirit, DataNucleus evaluates JPQL or JDOQL queries in memory. Of course, DataNucleus will leverage the capabilities of the data store as much as possible, and complement the missing query evaluation steps [Dat14c, Dat14b]. Due to this query shredding technique, DataNucleus is currently the only library among the evaluated ONMs that supports aggregate functions. Hibernate OGM has announced similar functionality for future releases [Red14a, Red14b] using the Teiid[1] engine.

Unfortunately, none of the analyzed Multi Data Store Mappers offers a **MapReduce API** so far. However, Kundera as well as Hibernate OGM have announced MapReduce support for future releases [Red14b].

Finally, some words on **transactions**. Using JPA or JDO Object-NoSQL Mappers, it is possible to define transactions. However, whether this is feasible depends on the underlying NoSQL data store, since many NoSQL data stores do not support transactions. Object mappers (ORMs as well as ONMs) normally use complex caching mechanisms. As long as the data is stored in the cache only, a transaction can be rolled back. Yet once the data has been flushed to the NoSQL data store, this is no longer possible (except with data stores which support ACID transactions, such as Neo4j or Infinispan).

4 Support for Schema Management with Object-NoSQL Mappers

Most NoSQL data stores are schema-less or schema-flexible. This offers great flexibility in the early stages of application development. Yet for long-term schema management, the tools currently provided by the vendors of NoSQL data stores are too rudimentary [KSS14]. Thus, any but the most basic data management tasks (such as persisting an object after renaming a property) usually require custom coding. However, Object-NoSQL Mappers also support certain schema management tasks. In the upcoming discussion we refer to the schema management requirements proposed in [SKS13] and [KSS14].

Schema definition Unlike relational databases, schema-less data stores typically do not manage a full-blown data dictionary. Yet even then, the application source code commonly contains class declarations. When developers use Object-NoSQL Mappers, the object mapper annotations not only declare the structure of persisted entities, but also the relationships between them. (Depending on the underlying NoSQL data store, the mapping strategy will vary.) So even when the NoSQL data store itself does not manage an explicit schema, the schema is nevertheless implicit in the class annotations in the application code.

[1] teiid.jboss.org

```
1  @Entity
2  public class Profile {
3      @Id
4      int profileID;
5      String firstname;
6      String lastname;
7
8      @Column(name="year")
9      int yearOfBirth;
10     String country;
11 }
```

(a) Renaming an attribute with `@Column`.

```
{
  "profileID"   : 4711,
  "firstname"   : "Miroslav",
  "lastname"    : "Klose",
  "year"        : 1978,
  "country"     : "DE"
}
```

(b) State of the (unchanged) persisted object in a document store.

Figure 6: Renaming attributes in Multi Data Store Mappers.

Validation To a certain degree, Object-NoSQL mappers also validate data against the implicit schema. As long as all access paths to the persisted data are using the same object mapper class definitions, the persisted data fits the class model. However, when the application code evolves, so does the class model. Thus, a persisted object may either fit the current or some historic class model, which motivates the next paragraph.

Schema evolution Schema evolution operations include adding, deleting, renaming, and updating entity classes. The latter covers all aspects of changing names or types of attributes as well as moving or copying attributes between entity classes. Such denormalization operations are essential in systems that do not support joins in query evaluation. Furthermore, relationships may be added or deleted, or the cardinality of relationships may be changed. In the following, we focus on these operations and their realization in Object-NoSQL Mappers in detail.

Adding an *attribute* works very well with Object-NoSQL Mappers. When loading a persisted object, the new attribute is added and initialized. Upon persisting the entity, the new attribute is persisted as well. This form of migrating data one entity at-a-time, at the time when it is loaded, is known as *lazy migration* [SKS13].

Removing a class member attribute from a class declaration results in *lazily deleting* the *attribute*, since it will no longer be loaded into the application space. Again, this only affects entities that are loaded at application run time, all other entities remain unchanged.

For *renaming attributes*, there are different strategies. In Multi Data Store Mappers, it is only possible to implement an aliasing approach using the `@Column` annotation [Jav09]. The new name is only available in the application space, while the persisted object itself is not changed. As an example let us rename the attribute `year` to `yearOfBirth` in our `Profile` class (c.f. Section 1, Figure 1). Figure 6 shows the JPA declaration `@Column` annotation in line 8 and the corresponding (unchanged) persisted object.

```
1  @Entity
2  public class Profile {
3      @Id
4      int profileID;
5      String firstname;
6      String lastname;
7
8      @AlsoLoad("year")
9      int yearOfBirth;
10     String country;
11 }
```

(a) Renaming an attribute with `@AlsoLoad`.

```
{
  "profileID"  : 4711,
  "firstname"  : "Miroslav",
  "lastname"   : "Klose",
  "yearOfBirth": 1978,
  "country"    : "DE"
}
```

(b) State of the (unchanged) persisted object in MongoDB after lazy migration.

Figure 7: Lazily renaming an attribute using Morphia annotations.

In contrast, the Single Data Store Mapper Morphia provides an annotation `@AlsoLoad` which lazily migrates persisted objects. Figure 7(a) shows a class declaration with the `@AlsoLoad` annotation in line 8 and the state of the object after persisting in Figure 7(b). Thus, on loading persisted objects with the now deprecated attribute name, the attribute name is changed. The change becomes permanent upon persisting the object in MongoDB.

None of the analyzed Object-NoSQL Mappers *copy* or *move attributes between* objects. A workaround is to use JPA's lifecycle annotations [Jav09]. For instance, methods annotated with `@PostLoad` are executed after the object has been loaded from the data store into the object mapper application context. Application developers can thus implement arbitrarily complex modification operations within these methods on a per-object basis.

We next consider operations beyond modifying single attributes. *Adding* a new *entity class* is straightforward. After *deleting entities* by removing the class declaration from the application source code, the entities are no longer accessible for the application. However, any persisted objects will remain in the NoSQL data store. Unfortunately, there is no systematic support for purging this data in the NoSQL data store in the Object-NoSQL Mapper libraries evaluated by us.

Renaming entity classes works with a similar aliasing approach like renaming an attribute, and is based on the `@Table` annotation [Jav09]. Again, the new name is used in the object-oriented application context and the old entity class name is used in the data store for existing entities, as well as when new objects are added. This is illustrated in Figure 8.

Adding and *deleting relationships* is similar to adding and deleting attributes in the analyzed Object-NoSQL Mappers. *Updating relationships*, i.e. changing the cardinality of the relationship, is not support by any of the evaluated mappers.

Data migration Apart from adding and deleting attributes or relationships w.r.t. single persisted objects, the analyzed Multi Data Store Mappers do not provide further data mi-

```
@Entity
@Table(name="Profile")
public class User {
    @Id
    int profileID;
    String firstname;
    ...
}
```

(a) The Java class declaration

Profile

profileID	firstname	lastname	year	country
4711	Miroslav	Klose	1978	DE
5000	Andre	Hahn	1990	DE

(b) State of the table (table name unchanged) in a column-family store after adding a new object.

Figure 8: Renaming entity classes with the `@Table` annotation.

gration operations. For renaming operations they do not change the data in the NoSQL data store. The Single Datastore Mapper Morphia at least is able to migrate lazily, e.g., in renaming attributes.

In general, life-cycle annotations like `@PostLoad` may be used to implement lazy and complex migration tasks on a per-object basis. Nevertheless, there is no systematic and well-principled support for automated data migration in the Object-NoSQL-Mappers studied here. According to [Red14b], Hibernate has a migration engine with support for lazy migration on its roadmap.

Schema versions Object-NoSQL Mappers can handle different variants of objects. For example, after adding an attribute, objects already persisted without this attribute can still be loaded. Likewise, after deleting an attribute, objects containing the deprecated attribute can be loaded. However, there is no explicit management of different schema versions, e.g., using version numbers.

Schema extraction If data was persisted without the support of an Object-NoSQL Mapper, it can still be interesting to extract the schema from the persisted data. Apart from gaining valuable insight into the data's structure, this would allow developers to conveniently access the data from Object-NoSQL Mapper libraries. In [KSS15] we sketch out this vision in greater detail and present a schema extraction algorithm.

Conclusion Object-NoSQL Mappers extend NoSQL data stores by implicit schema definition and implicit schema validation. Basic schema evolution operations, such as adding and deleting attributes or relationships (and renaming attributes in Morphia), are currently supported. However, when several schema evolution operations are to be applied, developers need to resort to custom code. A more general support for schema evolution in Object-NoSQL-Mappers is certainly desirable.

5 Performance of Object-NoSQL Mappers

In the decision which Object-NoSQL Mapper to use, their impact on the application runtime performance is a vital criterion. After all, the response time in interactive applications is crucial for the user experience. Therefore, two questions drive our experiments:

- Will an Object-NoSQL Mapper noticeably decrease runtime performance compared to the native API, as provided by the NoSQL data store vendor?
- Do different Object-NoSQL Mappers show different runtime performance?

For investigating these questions, we have generated synthetic data for our running example and we have further defined a set of queries, since we could not employ the YCSB benchmark [CST+10]: YCSB does not define an interesting data model with relationships between entities, and does not define any queries apart from basic CRUD operations.

5.1 Test Setup

We have experimentally evaluated the Object-NoSQL Mappers discussed previously against a range of NoSQL data stores. The version numbers for the ONMs are listed in Table 1. We ran each ONM against the following NoSQL data stores: MongoDB 2.4.6, CouchDB 1.0.1, Cassandra 2.0.5, and HBase 0.94.11, all running on Ubuntu 12.04, always provided that the ONM supports the system. The NoSQL data stores were used as-is, without any individual tuning.

All experiments were run on the same hardware with a typical NoSQL cluster environment consisting of commodity machines: Client and server processes ran on a single machine, each within the same local network. The Dell PowerEdge C6220 machines each have 2 Intel Xeon E5-2609 (4 Cores each), 32 GB RAM, and 4 x 1 TB SATA 7.2 k HDs. Since our focus was not to test the performance of the underlying NoSQL data stores, but the runtime overhead introduced by the Object-NoSQL Mappers, we consider it sufficient to evaluate the data stores in single-node configuration. The runtime performance was measured by profiling calls to ONM methods by manually injected statements.

5.2 Use Case and Synthetic Data

Figure 9 shows the class model on which we base our experiments. The model is an extension of our social network example and includes relationships with different types of functionalities. Due to the limitations regarding ElementCollections (c.f. Section 3), the relationships were implemented as one-to-many-relationships (c.f. Figure 5). This allows for a fair comparison of all ONMs. We generated the same test data for all experiments, ranging between 10^3 and 10^6 profiles. Each `Profile` has five `WallEntry` objects, each with two `Comments` on average. Further, each `Profile` has three `Chat` objects, and each `Chat` contains five `Message` objects on average.

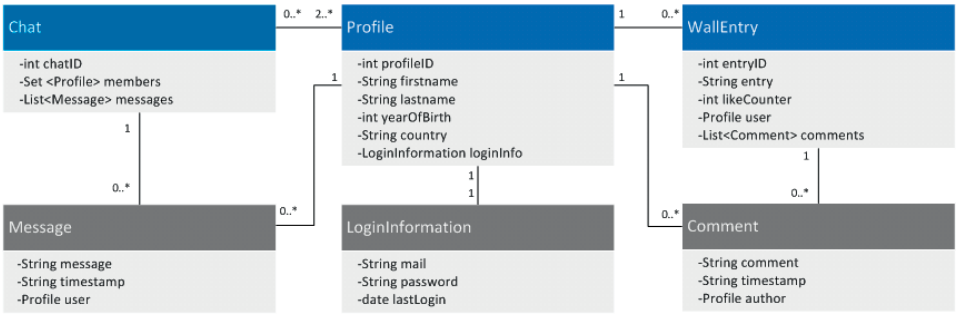


Figure 9: Class diagram for our application scenario.

Due to space limitations we can only present the highlights of our performance evaluation and refer to [Hau14] for the unabridged results. In the following, we only report the results for MongoDB, since it is the data store supported by all investigated mappers. Whenever the choice of NoSQL data store has a noticeable impact, we explicitly point this out.

5.3 Read Performance

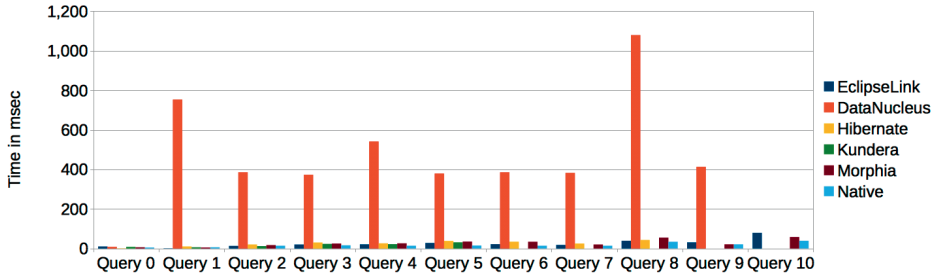
Appendix A lists the queries. Queries 0 through 9 are selections with varying predicates, e.g. atomic predicates with different data types, testing for equality and inequality, conjunction, disjunction, including basic text search functionality. Query 10 computes a join.

Differences in runtime overhead The runtime overhead of the object mappers on top of MongoDB[2] is shown in Figure 10. We refrain from listing the results for the other NoSQL data stores, since they do not contribute new insights. The runtimes in Figure 10 were collected by evaluating the queries over 10,000 Profiles. Some results are not shown, since not all ONMs support all queries.[3]

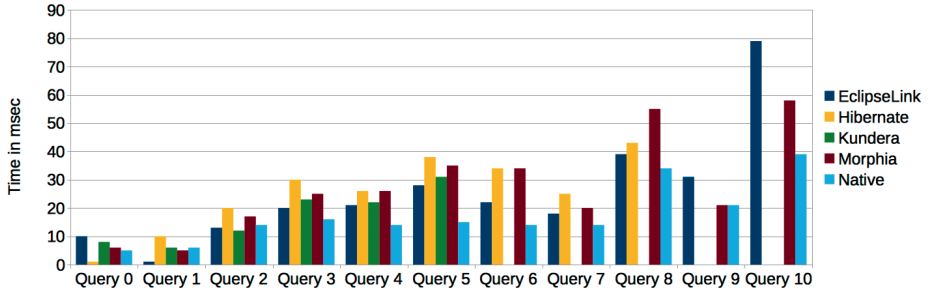
To our surprise, DataNucleus is significantly slower than its competitors on Queries 1 through 10, which all use JPQL syntax. In Figure 10(b) we omit DataNucleus to better show the differences between the remaining ONMs. The same effect occurs when using DataNucleus with other NoSQL data stores. It remains to be determined why the runtime overhead introduced by DataNucleus is considerably higher for JPQL queries. At this point it is merely a conjecture, yet DataNucleus is the only library under investigation that implements the same API for all supported NoSQL data stores, complementing missing features that are not provided by the data store inside the ONM library (c.f. Section 3.3). It may well be that this generality comes at the cost of a considerable performance overhead.

[2]The "native" access baseline for MongoDB is the MongoDB Query Builder API.

[3]DataNucleus does not support Query 10. Kundera returns the wrong number of results for Query 6 and the empty result for Queries 7-10 (for Queries 8-10 the reason is an incorrect implementation of the `LIKE` operator). Hibernate OGM throws exceptions for Queries 9 and 10, stating that they are too complex to be evaluated.



(a) Including DataNucleus.



(b) Excluding DataNucleus.

Figure 10: Query performance on 10,000 profiles (MongoDB).

Apart from DataNucleus, the outlier in this study, the overhead of ONMs for read operations is within about a factor of two when compared to the native access, the baseline in our evaluation. Thus, the other ONM libraries all show similar runtime performance.

Syntax sensitivity As an interesting finding, we point out the runtime differences between the two equivalent Queries 0 and 1. Both queries retrieve a single `Profile` based on the `profileID`. Query 0 uses the JPA interface method `find`, whereas Query 1 uses JPQL syntax. EclipseLink and Hibernate OGM differ in their runtime behavior in Figure 10(b). Hibernate is more efficient when using the `find` operator, whereas, surprisingly, EclipseLink is more efficient in using the JPQL interface. Thus, the query language is not as declarative as may be assumed, and the choice of query operator can have a noticeable impact on the runtime performance.

Scale up over larger datasets Figure 11 shows the scale up for Query 4 over larger data sets in MongoDB. Except for DataNucleus, the ONMs are about 1.4 to 2 times slower than access via the native API. The behavior for the other queries is similar.

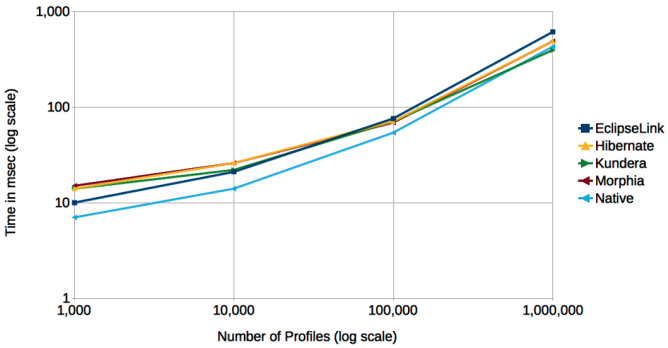


Figure 11: Scale up in evaluating Query 4 (MongoDB), not showing DataNucleus.

Matters of robustness and answer quality Unfortunately, the robustness and quality of query evaluation were not to our full satisfaction. For instance, Kundera returns correct results when processing 100,000 profiles with Query 4 on both MongoDB and Cassandra. Yet on Cassandra, the same query returns a wrong number of results when processing 1,000,000 profiles.

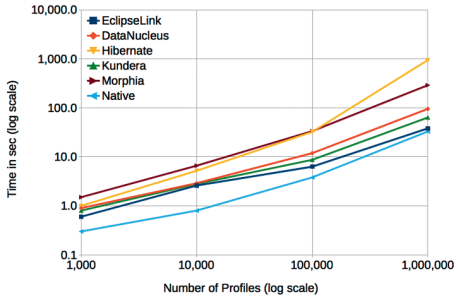
This is an alarming discovery, and shows that when choosing among ONM libraries, software architects need to test them for answer quality and reliability. Furthermore, quality assurance must go beyond simple unit tests, but be conducted on larger data sets, since some problems only reveal themselves when processing data at scale. To make matters worse, the correctness of answers depends on the underlying data store.

5.4 Write Performance

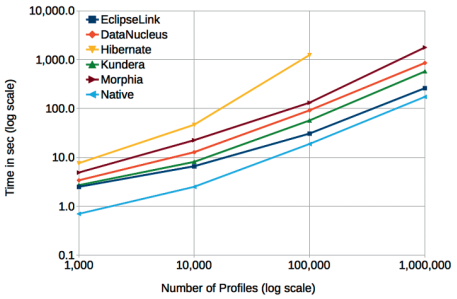
Next, we evaluate insert, update, and delete operations. It turns out that the runtime overhead for write operations is more noticeable than for read operations. Also, we now notice significant differences between libraries. Again, we only report our experiments on MongoDB, since the other NoSQL data stores do not contribute new insights.

Insertion We evaluate the insert performance for entities as well as for all relationship types of the class model shown in Figure 9. We are particularly interested in the results for inserting single entities and 1:1 relationships, as well as 1:N relationships. Figure 12(a) shows the scale up for inserting `Profiles` with `LoginInformation`. Because this is a 1:1 relationship, the matching data is stored within the same document. For 100,000 profiles, the slowest ONM is about 10 times slower than the fastest one, and about 15 times slower than access via the native API.

Figure 12(b) reports the result for inserts with a 1:N relationship, namely `Profile` objects with `WallEntrys`. The experiment for 1,000,000 `Profile` objects with Hibernate

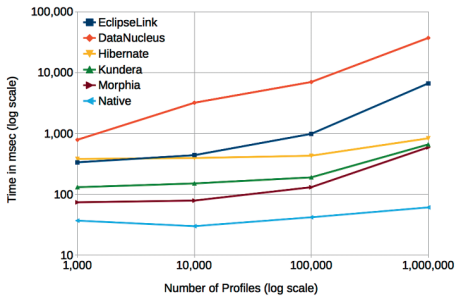


(a) Inserting Profiles.

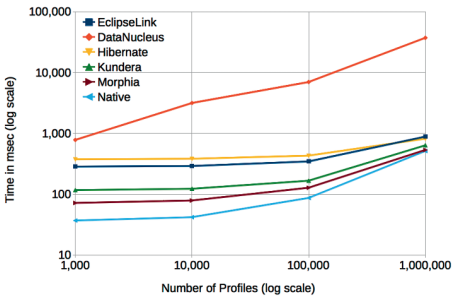


(b) Inserting 1:N relationships.

Figure 12: Scale up for insertion (MongoDB).



(a) Updating 25 Profiles.



(b) Deleting 25 Profiles.

Figure 13: Update and delete operations (MongoDB).

OGM was aborted after only 500,000 successful inserts within 10 hours. Obviously, the runtime differences between the ONMs are significant.

Updates and deletes For updates, we ran the following experiment. Query 4 returns 25 objects on our sample data, for all data sets of all sizes. These 25 objects are then updated within the application logic. Afterwards, the ONM generates the corresponding data store updates. We followed the same approach to test deletes. Figure 13(a) shows the runtime overhead for updates, while Figure 13(b) shows the results for deletes. Again, there is a significant gap between the native access and the mapper libraries. Moreover, the Object-NoSQL Mappers differ in the runtime overhead for updates and deletes.

Summary on writes In general, the effects on insert performance with data stores other than MongoDB are comparable, particularly the ranking of the different Object-NoSQL Mappers is very similar. However, the runtime differences between the slowest and the

fastest ONM and especially the runtime overhead using any ONM instead the native API depends strongly on the underlying NoSQL data store (see [Hau14] for details).

6 Summary

This paper gives an overview of the state-of-the-art in Java Object-NoSQL Mappers. We have studied very prominent, popular, and sophisticated libraries. Our analysis shows that it is safe to expect that a contemporary ONM provides basic CRUD operations.

However, the supported query languages differ greatly in their expressiveness. Which query operators are offered often depends on the capabilities of the underlying NoSQL data stores. This is a fundamental limitation for application portability.

During our experiments, we encountered some unexpected glitches: We ran into cases where query operators had not yet been implemented, even though the documentation describes them in full. Also, some query operators are not implemented with the semantics described in the documentation. As a consequence, developers must not rely on the documentation alone, and should show extra care when conducting their test cases.

Nevertheless, it is indisputable that application development with ONMs has its benefits:

- ONMs provide a (currently still restricted) vendor independent query language, one of the greatest drawbacks when working with NoSQL data stores.
- Most ONMs support several NoSQL (and relational) data stores within the same application (polyglot persistence).
- ONMs extend NoSQL data stores with a form of implicit schema definition. They perform basic schema validation, as well as some basic schema evolution operations: Adding and deleting attributes or relationships are commonly supported.

Our experiments reveal that in reading data, there is only a small gap between native access and the Object-NoSQL Mappers for the majority of the evaluated products. Yet in writing, object mappers introduce a significant overhead. This has to be considered in the decision whether to use a Object-NoSQL Mapper, and which library to choose in particular.

Despite their current limitations and drawbacks, Object-NoSQL Mapper may actually come to the rescue for many desperate application developers who find themselves struggling with schemaless and proprietary NoSQL data stores.

Exciting and new features, such as more powerful query engines, MapReduce support, and improved support for data migration have been announced. Thus, it will be interesting to watch the developments in the ONM market over time. Due to the increasing importance of scripting languages in web applications, it could be an interesting next step to evaluate the market for ONMs for languages like JavaScript, Ruby, or Python.

References

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC 2010, Indianapolis, Indiana, USA, June, 2010*, pages 143–154, 2010.

[Dat14a] DataNucleus. *DataNucleus AccessPlatform: Datastore Feature Support*, 2014. `www.datanucleus.org/products/accessplatform_4_0/datastores/datastore_features.html`.

[Dat14b] DataNucleus. *DataNucleus AccessPlatform: JPA : JDOQL Queries*, 2014. `www.datanucleus.org/products/accessplatform_4_0/jdo/jdoql.html`.

[Dat14c] DataNucleus. *DataNucleus AccessPlatform: JPA : JPQL Queries*, 2014. `www.datanucleus.org/products/datanucleus/jpa/jpql.html`.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI, San Francisco, California, USA, December, 2004*, pages 137–150, 2004.

[Ecl14] EclipseLink. *Understanding EclipseLink 2.6*, 2014. `http://www.eclipse.org/eclipselink/documentation/2.6/eclipselink_otlcg.pdf`.

[Ecm13] Ecma International. The JSON Data Interchange Format, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[EFH+11] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer, and Markus Brückner. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Carl Hanser Verlag, 2011.

[Hau14] Thomas Hauf. Vergleich von Object to NoSQL Mappern. Master's thesis, University of Applied Sciences Darmstadt, 2014.

[Imp14a] Impetus. *Kundera JPQL*, 2014. `https://github.com/impetus-opensource/Kundera/wiki/JPQL`.

[Imp14b] Impetus. *Kundera Wiki*, 2014. `https://github.com/impetus-opensource/Kundera/wiki`.

[Jav09] Java Persistence 2.0 Expert Group. *JSR 317: Java Persistence 2.0*, 2009.

[KSS14] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Datenbanken ohne Schema? Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schemaflexiblen NoSQL-Datenbanksystemen. *Datenbank-Spektrum*, 14(2):119–129, 2014.

[KSS15] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Proc. 16. BTW, March, 2015 Hamburg, Germany*, 2015.

[Lig10] Sam Lightstone. *Making it Big in Software*. Prentice Hall, 2010.

[Mon14] MongoDB. *Morphia Wiki*, 2014. `https://github.com/mongodb/morphia/wiki`.

[MW12] Bernd Müller and Harald Wehr. *Java Persistence API 2: Hibernate, EclipseLink, OpenJPA und Erweiterungen*. Carl Hanser Verlag, 2012.

[Red14a] Red Hat. *Hibernate OGM Reference Guide 4.1.0*, 2014. http://docs.jboss.org/hibernate/ogm/4.1/reference/en-US/pdf/hibernate_ogm_reference.pdf.

[Red14b] Red Hat. Hibernate OGM Roadmap, 2014. http://hibernate.org/ogm/roadmap/.

[Red14c] Red Hat. *Hibernate Search: Apache Lucene Integration Reference Guide*, 2014. http://docs.jboss.org/hibernate/stable/search/reference/en-US/pdf/hibernate_search_reference.pdf.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison Wesley, 2012.

[SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. 14th International Symposium on Database Programming Languages (DBPL 2013), August, 2013, Riva del Garda, Trento, Italy*, 2013.

[Tiw13] Shashank Tiwari. *Professional NoSQL*. O'Reilly, 2013.

A Appendix: Queries used in our experiments

0. `find( Profile,` *intvalue* `)`
1. `SELECT p FROM Profile p`
   `WHERE p.profileID=` *intvalue*
2. `SELECT p FROM Profile p`
   `WHERE p.firstname=` *stringvalue* `AND p.lastname=` *stringvalue*
3. `SELECT p FROM Profile p`
   `WHERE p.yearOfBirth=` *intvalue*
4. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue*
5. `SELECT p FROM Profile p`
   `WHERE p.yearOfBirth>=` *intvalue* `AND p.yearOfBirth<=` *intvalue*
6. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue* `AND`
   `p.firstname <>` *stringvalue* `AND p.firstname <>` *stringvalue*
7. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue* `AND`
   `( p.firstname =` *stringvalue* `OR p.firstname =` *stringvalue* `)`
8. `SELECT w FROM WallEntry w`
   `WHERE w.likeCounter >=` *intvalue* `AND w.entry LIKE %`*stringvalue*`%`
9. `SELECT p FROM Profile p`
   `WHERE p.loginInfo.mail LIKE %@` *stringvalue* `AND`
   `p.country IN (` *stringvalue* `,` *stringvalue* `)`
10. `SELECT w FROM WallEntry w JOIN w.comments c`
    `WHERE c.comment LIKE %` *stringvalue*`% AND w.entry LIKE %` *stringvalue*`%`

Semantic Data Exchange in e-Navigation

Mazen Salous[1], Daniela Nicklas[1], Andre Bolles[1], and Henning Mextorf[2]

[1]Bereich Verkehr, OFFIS Institut für Informatik, OFFIS e.V., Escherweg 2, 26121 Oldenburg
[1]mazen.salous@offis.de, nicklas@offis.de, andre.bolles@offis.de
[2]Raytheon Anschütz GmbH, ZeyestraÃ§e 16, 24106 Kiel
[2]Henning_ Mextorf@raykiel.com

Abstract: E-Navigation aims to enhance the safety and security of maritime activities by exchanging, integrating and analyzing the data between and in ship-side and shore-side systems. In this paper, we present the main characteristics of our semantic data exchange concept developed in the COSINUS project. In addition to the main task of exchanging data written in a variety of formats between on-board and onshore systems, each semantic data processor, which is related to a maritime system like ECDIS (Electronic Chart Display and Information System) or VTS (Vessel Traffic Service), defines a context model, which itself concludes information from the transient streams of data, registers the current state and the history of the ships, assigns some important meta-data such as data quality information and applies semantic compression by sending the interesting patterns only. Therefore, the context models in our data exchange represent cooperatively the navigation situation and this enables cooperative control and monitoring on such navigation and ensures the integration between different marine data sources.

Keywords

e-Navigation, Maritime, VTS (Vessel Traffic Service), ECDIS (Electronic Chart Display and Information System), Semantic Data

1 Introduction

Waterways in the German Bight are important economic infrastructures. Safe maritime routes in terms of avoiding accidents and collisions are particularly important for Germany for the international exchange of goods, taking into account that the vast majority of German goods transfers (276 million tones of goods were handled in 2010) are handled by the sea [Win11]. Increasing ship traffic, technical and human errors (often caused by insufficient situation awareness) and extreme weather events (storms, unusual sea conditions) affect clearly the traffic management on the shipping lanes and may cause a wide range of hazards for the ship safety. E-Navigation requires integrated maritime traffic control and

safety systems in order to increase the efficiency of maritime transport and the protection of the marine and coastal environment.
More challenges should be reckoned within the German Bight because of the possible conflicts between shipping and other usages of the maritime space like construction of offshore wind farms. According to the Wind Energy Agency Bremerhaven (Windenergie Agentur Bremerhaven)[1], it is planned to construct approximately 4538 plants in 87 wind farms in the German Bight in the next few years, this will restrict the existing shipping routes and increase the risks of possible collisions: ship/ship and ship/wind parks.
The safety of maritime systems, the integration, the improved communication and the cooperation between actors involved in dangerous situations are central objectives of the e-navigation strategy of the International Maritime Organization (IMO) [IMO13]. The National Master Plan Maritime Technologies (NMMT)[2] and the research programme "Maritime next-generation technologies(2011-2015)" [3] have been incorporated for the achievment of these objectives. The International Maritime Organization (IMO) has called for a cooperative communication management between ship-side systems and shore-side systems in order to prevent accidents and increase the safety. The objective of research and development must be an increased degree of safety in maritime traffic. This can be supported by exchanging of useful information between ship-side systems and shore-side systems.
Therefore, the research project COSINUS[4][BB14] examines the integration of navigation systems on ship and shore sides. The main contribution is to produce a comprehensive situational awareness on board a vessel as well as in land-based vessel traffic services (VTS) centers, in order to assist the mariners in the prevention of collisions and groundings and thus to increase the safety of shipping despite the continuous increasing of traffic density.
In this paper, we present the architecture of the semantic data exchange in COSINUS project and we go through the details of two use-cases based on such semantic data exchange. After this general introduction, the rest of the paper will go as the following: in the related work section, we try to point out to the level of knowledge and development achieved in data exchange and e-navigation domain, and the next section will highlight the contribution of our project COSINUS in e-navigation. Then we present the architecture of the semantic data exchange in COSINUS. In the implementation section, we will introduce the complex event processing system we are using in COSINUS and we describe our current progress in the project and the first demonstration we have already done. After that, we go into some details with two use-cases of semantic data exchange and we summarize in the conclusion.

[1] http://www.wab.net/images/stories/PDF/offshore/WAB_Offshoretabelle_A4_Nord-OstseeDE_06-2011.pdf

[2] http://www.nmmt.de

[3] http://ec.europa.eu/research/biotechnology/eu-us-task-force/pdf/strategic-plan-2011-2015.pdf

[4] http://www.offis.de/f_e_bereiche/verkehr/projekt/projekte/cosinus.html

2 Related Work

We will refer in this section to the state-of-the-art projects, which have contributed in data exchange in the e-Navigation domain. These projects have been mentioned in our COSINUS project paper [BB14]. Not all the projects in e-navigation have considered the integration of integrated bridge systems (INS) and VTS systems, for example: The project BaSSy [5]- Baltic Sea Safety and the InterReg project EfficenSea [6] focused on analyzing AIS information and radar images and contributed to risk identification algorithms for VTS systems regardless the data exchange and integration between INS and VTS systems. On the other hand, there are projects have been contributing in data exchange and integration between maritime systems, like ACCSEAS [7] and MonaLisa I + II [8].
ACCSEAS focuses on the development of prototypical e-navigation services and the development of a testbed in the North Sea region to demonstrate the prototype servies. One of the prototypes demonstrates the functionality of route exchange between vessels and VTS systems, and between vessels themselves.
Monalisa I + II focuses also on data exchange in the term of route exchange from ship to shore and ship to ship. The difference between COSINUS and Monalisa is in the architectural design, while Monalisa project assumes a central administrative entity that organizes route planning among different participants in a specific area at sea, ours avoids the need for such a central organizational body, but relies on the judgement of all parties involved (ship masters and VTS operators) instead [BB14].

3 COSINUS with the current e-Navigation Systems

As we mentioned above, the COSINUS project aims to enhance the existing navigational systems by ensuring the data integration onshore and on board. Today, the only communication between ship-side and shore-side is done by the staff in the control centre on shore and the mariners on the vessels. The contribution of COSINUS is the technical support for enabling the cooperation between the both sides.
In today's systems, some data fusion and integrity checks are carried out, e.g. the fusion of AIS (Automatic Identification System) and radar targets in ECDIS. COSINUS aims at the integration of such maritime traffic safety systems on board and on shore like ECDIS and VTS. Additionally, novel concepts for the presentation of enhanced data to the operator and operation of new tools and services as well as decentralized data capturing, processing and storage are investigated.
The data in land-based information systems will be visualized in a way that a complete overview over the traffic situation is given in order to support the navigational operation of the vessel. This includes the presentation of the shared routes and manoeuver plans as

[5] http://meeting.helcom.fi/c/document_library/get_file?folderId=75266&name=DLFE-30282.pdf
[6] http://www.efficiensea.org/
[7] http://www.accseas.eu/
[8] http://monalisaproject.eu/

well as the operational interface to the VTS operator. The goal is to establish a cooperative and task-based picture, which offers a dynamically enhanced view for the bridge crew going beyond traditional ship-based sensor information like the own-ship radar or AIS. The systems, which offer a cognitive ability, are of great interest whenever it comes to critical situations where an enhanced situation awareness is required. This could be information of a higher order, e.g. an indication of the actual anticipated danger level.

4 Semantic Data Exchange Architecture

Many different streams of data, in terms of different formats (syntax) and semantics, are published continuously by ship-side and shore-side systems. Advanced complex event processing systems will undertake the tasks of data exchange, data fusion, data provenance, data integration, data mining, data cleansing and data quality processing.
In figure 1 you can see the abstract design architecture of the semantic processor. There is a semantic processor for each system on-board and onshore. Each semantic processor is responsible of organizing the traffic of different streams in both directions, i.e. the messages sent or received by the related navigation system (ECDIS, VTS).

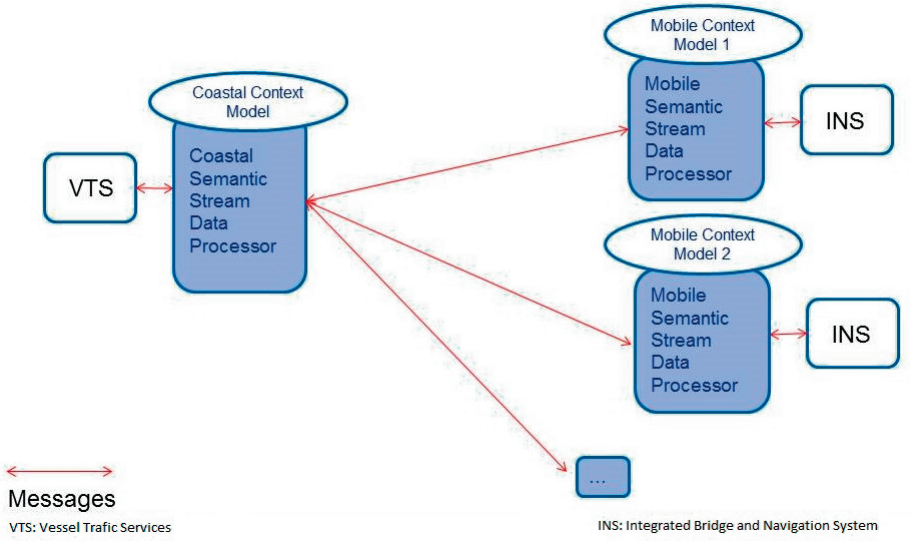


Figure 1: Semantic Data Exchange Architecture

4.1 Coastal and Mobile Semantic Stream Data Processor

4.1.1 Definitions

Definition 1. *The Coastal Semantic Stream Data Processor is an onshore complex event processing system responsible of organizing the traffic of messages (represented in a variety of different formats) sent or received by the VTS.*

Definition 2. *The Mobile (ship) Semantic Stream Data Processor is an on-ship complex event processing system responsible of organizing the traffic of messages (represented in a variety of different formats) sent or received by the ECDIS .*

4.1.2 Functions

The VTS System in use supports the IVEF format [9] among others, such as NMEA and IEC61174. But it will receive messages written in any format (NMEA, IEC61174..) sent by the ECDIS on the ship side via the data processors. On the other hand, the ECDIS will have to receive IVEF messages from the VTS system. Therefore the coastal and mobile data processors will have to convert between the different formats, and may leverage the existing systems by providing them with missing data. Moreover, they can conclude useful information from the passed-through messages and then such information can be used to enrich the related context model. More details about the coastal and mobile context models are available in section 4.2. The main functions of the coastal and mobile semantic transformers can be summarized in the following:

- *Technical (non-semantic) Function:* Conversion (IVEF <->NMEA, IVEF <->JSON, IVEF <->IEC61174) and/or forwarding.
- *Semantic Function:* Enriching the coastal and mobile context models by extracting useful information from the transient (passed-through) messages.

4.2 Context Models

The context models contribute in ensuring a **cooperative control and monitoring** on the navigation situation.
The context model stores a mid-term memory / history of a semantic processor, e.g. one hour sliding time-window (can be specified). The context model will be built up continuously by interpreting the passed-through messages. That means, while the semantic processor is converting between standards and forwarding the messages, the related context model, in parallel, will be enriched with useful information concluded from such transient messages, like the detection of inconsistent information between the ship-side and the shore-side.

[9] `http://openivef.org/`

4.2.1 Mobile Context Model

The mobile context model is an abstract description of the navigation process from the vessel's point of view. Every useful result extracted from the data processing on the transient data should be registered in the related mobile context model. Therefore, if the data cleansing component, for example, detects some outliers and ignores them from the streams, the outliers should be registered.
Moreover, the mobile context model will ensure data traceability by storing a mapping history between the original data and the converted data.
A lot of useful applications can be done upon such context model, for example: When entering the range of a VTS, a ship can send some history of it's context, then the VTS system will be able to analyze such history data and predict the behavior of the new joined ship.

4.2.2 Coastal Context Model

As a counterpart to the mobile context model, the coastal context model stores traffic information coming from all available vessels and coastal surveillance stations. This monitoring includes not only the ships sensed by the VTS system directly, but also those apart from the coverage area of the VTS and their information were received via the semantic processors.
Using this context model, we can also rate the quality of data (e.g., completeness, accuracy) considering the availability of many data sources. Moreover, this coastal context model, as a leader one, should have suitable representations of the mobile context models of the communication partners (vessels). Then, a snapshot on this context model at time t will be useful to check some time-relevant concepts like the range: what could a ship report at time t.

5 Implementation

We have already implemented the data exchange between the ship-side and shore-side. The shore side system (VTS) works with IVEF format. It publishes and is able to receive IVEF messages containing tracks and/or routes.
On the other hand, the ship side system, ECDIS, works with NMEA-0183 format [10], JSON format and may work also with IEC-61174 format in the future. It publishes tracks in NMEA format, the user can import a route written in JSON and the ECDIS can then publish this route.
We have developed protocol handlers for each format in the Complex Event Processing System Odysseus[11][AN12]. Thus, Odysseus can receive the messages, written in any format, parse them and convert them into the counterpart format.
For such data exchange, queries will be prepared in order to receive the data from a mar-

[10] http://www.nmea.de/nmea0183datensaetze.html#gga
[11] http://odysseus.informatik.uni-oldenburg.de/

itime system, parse them using the protocol handler, and then these data will be represented in the internal format of Odysseus, so that we can use arbitrary algorithms like Kalman-Filter for further processing. Finally, a converter query operator is used to convert into the destination format.

First Demonstration

The first demonstration scenario represents a vertical prototype for data exchange between INS and VTS. In this scenario, we receive simulated target data from OFFIS' maritime traffic simulation (MTS)[DH14], and real target data from a sensor box (NaviBox) equipped with radar and AIS receivers also provided by OFFIS. Both targets, real and simulated, are received by Odysseus as NMEA sentences. Odysseus forwards them to the ECDIS and converts them into IVEF before sending them to the VTS. This vertical prototype was used to generate a consistent situational awareness on both systems using simple data such as target data.
The following figure represents the work-flow of operators in the complex event processing query of this scenario. The "MetadataCreationPO" and "SystemTime" are for preparing meta-data inside Odysseus during processing, while the other operators represent the sequence of processing done in this scenario, beginning from receiving the data from MTS and the NaviBox, then converting into IVEF using the "IvefNmeaConverter" and merging the results using the "Merge" operator, and at the end the "TCP-Server" will establish a TCP connection with the VTS system to send it the resulted IVEF data.

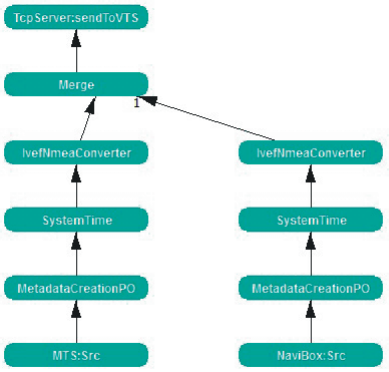


Figure 2: Query Plan: NMEA to IVEF

6 Semantic Data Exchange Features

The aspects, which have been already mentioned, give some details about the basic technical background to be considered when thinking about data exchange between maritime systems. However, data exchange will also allow us to generate high value information when taking the semantic of data into account. This section will present two examples we will address within COSINUS project for the semantic high value data processing. First, taking the semantic of data into account will allow us to compress the data to be exchanged, which is essential due to the low bandwidth available on the vessels. Second, having information from different information sources available will allow us and also enforce us to provide information about data quality to the operators and mariners.

6.1 Semantic Compression

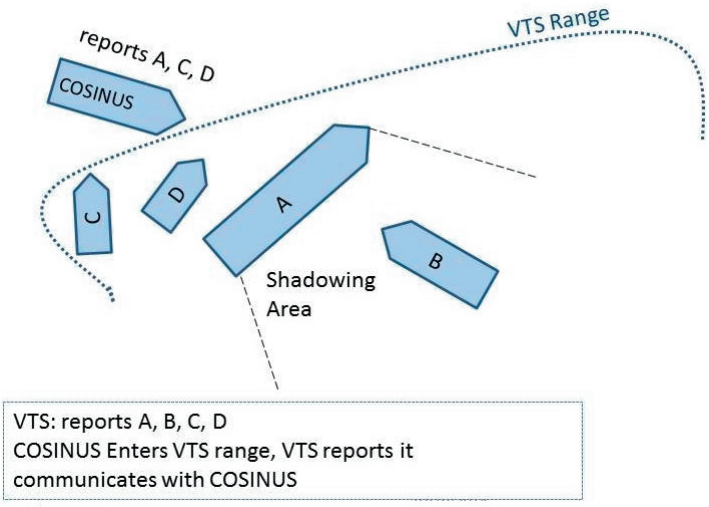


Figure 3: Semantic Compression

The semantic compression concept means to compress the data to be exchanged based on its semantic. We can do such semantic compression by detecting the interesting patterns and send only them instead of sending the whole data, or we can formulate a semantic function and send it's parameters instead, then the receiver side can use the semantic function to calculate the data. We will discuss here a use-case of semantic compression based on detecting the interesting patterns.
Considering the radar shadowing situation as an example, the ship "B" is detected as an interesting pattern for the ship "COSINUS", while the ships "C" and "D" are considered uninteresting patterns for "COSINUS", because the context models can know from the

ranges and the shadowing areas of the ships that "C" and "D" can be detected by "COSINUS"' radar sensor, while "B" cannot be detected because it is located in the shadowing area of "A".
The availability of many sources for the same data in e-navigation can contribute clearly in the reliability. However, we have to compress such redundant and/or unnecessary data based on the semantics of the data in order to enhance the efficiency in data transmission. The authors of [CC08] depend on identifying the interesting events occurring in an unbounded stream to make the compression. Similarly, based on the well-defined semantics, stored in the context models, we can detect the interesting patterns from the very huge amount of data streams and send only them. This semantic compression will increase the transmission efficiency clearly because we will send only a small set of interesting patterns instead of sending every available tuple of data published by a marine system.
According to such detection of interesting patterns, the semantic processors will pass only the interested pattern "B", and avoid sending the others. In parallel, the related context model of the semantic data processor will be enriched with information that tells that an interesting pattern was detected in a shadowing area. Such information in the context model can be useful later for several purposes as well. A useful reaction can be taken on COSINUS as a response of such data exchange, for example modifying the planned route a little bit to avoid a possible conflict with "B".

6.2 Data Quality Assessment (Completeness Example)

We should take into account that the continuous and high rate data streams in e-navigation should not only be converted between standards and exchanged, but also checked regularly in order to adjust or annotate the quality of such data. Data of low quality, which does not exceed some predefined thresholds, could be then ignored, annotated and/or cleansed.
Based on the spatio-temporal characteristics of the incoming data, we can conclude different data qualities in order to annotate them or may be to react with a suitable data cleansing.
Different types of data qualities can be extracted from the continuous transient streams. In this section, we will present a use-case/scenario for the completeness as a data quality aspect, which can be determined by our semantic data processors.
The main key for the data assessment in e-navigation is the availability of many sources for the same data. We can use the available replicas, which have the same semantic but may be written in different formats, in order to rate the completeness of data published by different sources. Thus, using the redundant replicas, we do not have to propose a solution like adapting the smoothing window size based on statistical samples [JF06], but it is enough to detect such missing data, register the detected information in the context model, and simply use another replica has the same semantic to ensure the completeness of the delivered data stream.
Recall the radar shadowing scenario, the VTS system can detect "B", while it cannot be detected by "COSINUS" because "B" is located in the shadowing area of "A". Therefore, the semantic processors will pass the information of "B" to "COSINUS" to ensure the

completeness, and "COSINUS" will know from the annotation registered in the context models that the reason for the inability to detect "B" itself is its location inside the shadowing area caused by "A". Having this information it can predict the time for "B" to leave the shadowing area. Thus, "COSINUS" will know that there is no problem in its radar, if "B" is re-detected again at that predicted time.

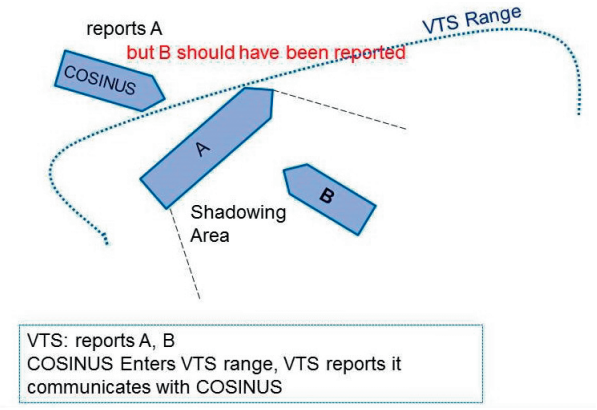


Figure 4: Completeness annotation and ensuring

7 Conclusion

We have presented in this paper our architecture in semantic data exchange in e-navigation. We are applying this semantic data exchange in COSINUS using the complex event processing system, Odysseus, and the maritime systems of our partners: The VTS system from SIGNALES and ECDIS from Raytheon.

We define context models to enable the cooperative control and monitoring on the e-navigation onshore and on-board. Based on the semantics we conclude from the data streams and register in the context models, we can rate the date qualities, apply suitable semantic compression, detect possible outliers and may apply suitable reaction such as data cleansing. For the future work, we can handle the contradictories coming from different data sources for the same target by applying data fusion on them, however, such contradictories should not cause that noteworthy effects when the used sensors and radars are with high qualities. The work presented in this paper has been done in the project COSINUS that is funded by the Federal Ministry of Economics and Energy under the support code 03SX367D.

References

[AN12] J. Geesen D. Grawunder M. Michelsen T. Appelrath, H. and D. Nicklas. Odysseus - A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In *DEBS '12 Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 367–368, 2012.

[BB14] Hahn A. Braun M. Rohde S. Gluch M. Bolles, A. and K. Benedict. COSINUS - Cooperative Vessel Guidance for Nautical Safety. In *The 10th International Symposium ISIS 2014 "Integrated Ship's Information Systems"*, 2014.

[CC08] A. Cuzzocrea and S. Chakravarthy. Event-Based Compression and Mining of Data Streams. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 670–681, 2008.

[DH14] C. Dibbern and A. Hahn. Maritime Traffic Co-Simulation For Analyses of Maritime Systems. In *28th European Conference on Modelling and Simulation ECMS 2014, At Brescia, Italy*, 2014.

[IMO13] Contribution from the International Maritime Organization (IMO) to the Secretary-General's report for the 2013 Annual Ministerial Review on Science, technology and innovation, and the potential of culture, for promoting sustainable development and achieving the Millenium Development Goals, 2013.

[JF06] R. Garofalakis M. Jeffery, S. and J. Franklin, M. Adaptive Cleaning for RFID Data Streams. In *VLDB '06 Proceedings of the 32nd international conference on Very large data bases*, pages 163–174, 2006.

[Win11] H. Winter. Seeverkehr 2010. Technical report, Statistisches Bundesamt, Wiesbaden., 2011.

Preference Analytics in EXASolution

Stefan Mandl, Oleksandr Kozachuk
EXASOL AG
Neumeyerstraße 24
90411 Nürnberg, Germany
firstname.lastname@exasol.com

Markus Endres, Werner Kießling
University of Augsburg
Universitätsstr. 6a
86159 Augsburg, Germany
lastname@informatik.uni-augsburg.de

Abstract: Skyline queries and the more general concept of preferences are well-known in the database community and there are many academic approaches for the computation of the best-matching objects. Furthermore, data analytics and multi-criteria optimization play an important role in Business Intelligence where it facilitates optimal decision making. *Preference Analytics* is the combination of preferences and data analytics. SKYLINE is EXASOL's implementation of Preference Analytics in its commercial database management system EXASOLUTION. In this paper, we present SKYLINE from a user's perspective, describe algorithmic design decisions, and discuss its implementation in a distributed and parallel environment. The paper closes with an empirical evaluation of the system based on a number of preference queries over the TPC-H dataset using different scale factors.

1 Introduction

The Skyline operator [BKS01] and the more general concept of preference database queries [SKP11, KEW11] has emerged as an important summarization technique for multi-dimensional datasets. The popularity of preferences is mainly due to their applicability for decision making applications [GRB11]. In such applications, the database tuples represent a set of multi-dimensional data objects, and the *Skyline*, also named *Pareto Set*, contains those objects that are the best trade-offs between different attributes, i.e. those objects that are not dominated by any others. An object p having d attributes dominates an object q, if p is strictly better than q in at least one attribute and not worse than q in all other attributes, for a defined comparison function. Preferences enable users to specify a pattern that describes the type of information he is searching for. Since preferences express soft constraints, the most similar data will be returned when no data exactly matches that pattern (Best-Matches Only set). From this point of view, preference database queries are an effective method to reduce very large datasets to a small set of highly interesting results.

EXASolution is a high performance parallel and distributed in-memory engine for data analytics and data warehousing produced by EXASOL AG[1]. It is the first commercial system which incorporates the functionality of Skylines for *Preference Analytics*. Subsequently, we show an example of a preference query using the novel SKYLINE syntax for Preference Analytics in EXASolution (see Section 3.1).

[1] http://www.exasol.com

Example 1. *Consider the task of finding the best funds in the market – a daily problem for financial investors. Assume one wants to invest in the fund that provides the* highest return *and the* lowest risk*. Unfortunately, investment funds do not work that way: the more conservative a fund, typically the lower its risk, but also its return value. Figure 1 presents a sample dataset of funds in the market and the Standard & Poor's 500 (S&P 500), which is a stock market index based on the market capitalizations of 500 large companies. So, there is no right answer as to which fund to select.*

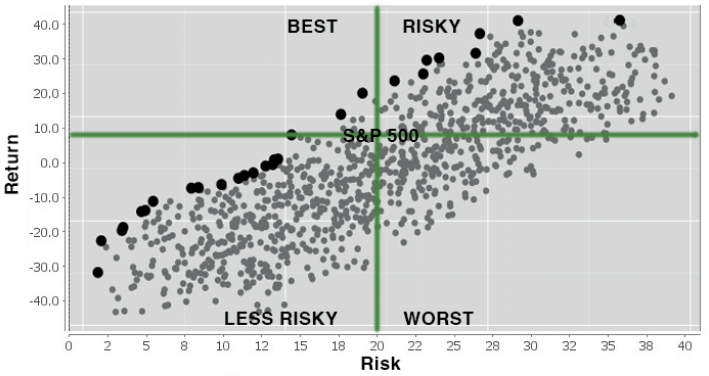


Figure 1: Example of the Fund Universe Comparison graph (*http://www.fundreveal.com*) and the best-matches only set (bold points).

However, using SKYLINE *you can tell the system that this is your preference and it will provide you with a shortlist of candidates – effectively eliminating all objects that are worse in every respect than competitors. In Figure 1 the bold points represent the optimal subset regarding the subsequent preference query:*

```
SELECT * FROM funds
PREFERRING HIGH return PLUS LOW risk;
```

Figure 2: Simple preference query to find the *best* fund.

The ***PREFERRING*** *keyword introduces a preference. The connection of two preferences by* ***PLUS*** *is the Pareto composition and states the equal importance of preferences. The keywords* ***HIGH*** *and* ***LOW*** *define whether an expression should have* high *or* low *values. The query selects all objects which are not dominated by any other objects in all dimensions. From this Skyline set one can now make the final decision, thereby weighing the personal preferences for a more conservative or more risky fund.*

Preference Analytics addresses the fundamental problems of traditional data mining approaches and multi-criteria decision making, where the ever-increasing volumes of data and multiplicity of variables mean that sorting, filtering and weighting lead to sub-optimal analyses. This capability is unique to EXASolution and was developed to help clients with more sophisticated analytic and data mining requirements. Potential application areas include segmentation, next best offer, fraud and portfolio analysis, etc.

The remainder of this paper is organized as follows: In Section 2 we discuss some related work. In Section 3 we present *Preference Analytics* including a parallel and distributed Skyline algorithm used by SKYLINE in EXASolution. Afterward we show elaborate examples of Preference Analytics in Section 4. We conduct a performance evaluation on the TPC-H benchmark dataset in Section 5. Section 6 contains our concluding remarks.

2 Related Work

Preferences in databases—as shown by a recent survey [SKP11]—are a well established framework to create personalized information systems. By using well designed preference models, users can be provided with just the information they need, thereby overcoming the dreaded empty result set and flooding effect [KEW11]. Traditional database engines or query languages do not support preference queries. However, in the last decade some extensions to the SQL language have been proposed.

Börzsönyi et al. [BKS01] introduced the **SKYLINE OF** clause with its basic form

```
SELECT ... FROM ... WHERE ...
  SKYLINE OF [DISTINCT] A1 [min | max | diff], ...,
                           An [min | max | diff];
```

The columns `A1, ..., An` are the attributes over which the preferences apply. Their domains must have a natural total ordering, such as integers. The directives **min** and **max** specify whether one prefers low or high values, respectively. The **diff** states that one wants to retain the best choices with respect to each distinct value of that attribute. Unfortunately, this approach is restricted to **min, max** and **diff**, such that complex preference queries are not possible.

Chaudhuri et al. show through an implementation in the commercial database product Microsoft SQL Server 2005 (Beta version) that Skyline queries can substantially benefit from cardinality estimation techniques [CDK06, End14]. However, their Skyline implementation never found its way into any product release of MS SQL Server. Eder and Fang [EF09] implemented the original Skyline syntax in PostgreSQL in order to show the integration of the Skyline operator with other relational operators, thus more sophisticated queries can be constructed. This Skyline implementation never became an integral part of the PostgreSQL database system.

A more general approach of Skylines are *preference queries*, which allow a more detailed specification of user wishes. *Preference SQL* is an extension to SQL to specify such arbitrary user preferences [KEW11]. Preferences in this sense are strict partial orders and a preference query returns the maximal elements according to this order, also called the *Best-Matches Only set* (BMO-set) [Kie02]. The basic form of a PreferenceSQL query is as follows:

```
SELECT ... FROM ... WHERE ...
PREFERRING A1 <preference_constructor> [ AND | PRIOR TO | RANK | DUAL ]
           ...                         [ AND | PRIOR TO | RANK | DUAL ]
           An <preference_constructor>;
```

The **PREFERRING** keyword introduces a *preference*. The connection of two preferences by **AND** is the *Pareto composition* and states the equal importance of preferences. If we restrict the attention to LOWEST (**min**) / HIGHEST (**max**) as input preferences, then Pareto preference queries coincide with the traditional *Skyline queries* above.

Arvanitis and Koutrika [AK12] pushed preference query processing closer to the database management system (DBMS) and explained how to integrate the preference operator and its optimization into a DBMS. Levandoski et al. [LEMK13] presented FlexPref, a general framework for extensible preference evaluation. FlexPref is implemented in the query processor of PostgreSQL, and supports various preference evaluation methods. Although FlexPref was integrated into PostgreSQL, it has never been an inherent part of it. Golfarelli et al. [GRB11] followed the basic principles of [Kie02] and developed myOLAP, an approach to express and evaluate OLAP preferences.

Preference query processing received enormous attention in single-database environments, i.e., in a centralized setup, cf. [CCM13], as well as in multi-core environments [PKP+09, SLB10, LVDN14, EK14]. As nowadays data are increasingly stored and processed in a distributed way, Skyline processing over distributed data has attracted much attention recently. EXASolution is a highly distributed database management system and therefore novel algorithms have to be developed for Preference Analytics. We give a short overview on existing approaches on distributed Skyline computation.

Balke et al. [BGZ04] proposed the first distributed Skyline algorithm, by considering the underlying relation is vertically partitioned between local servers, i.e., each server keeps only one attribute of the relation. A vertical partitioning is also used in [TBPY13], where a general solution for arbitrary dimensionality is proposed. In contrast, Wu et al. [WZF+06] assume a constrained horizontal partitioning. They used a CAN overlay to create grid partitions, each assigned to a processor. The main disadvantage of this approach is its low parallelism and that processors exchange the entire Skyline which floods the network. Zhu et al. [YSZ09] studied the distributed Skyline query when the underlying dataset is horizontally partitioned onto a set of local servers. Their algorithm can solve the problem with low bandwidth consumption. In [RJVDN09] the authors propose AGiDS, a framework for efficient Skyline processing over distributed data. Thereby a grid-based data summary is used to reduce the amount of transferred data. The main focus in the related work on distributed Skyline computation is on highly distributed systems, such as P2P systems, where each server stores a fraction of the available data, e.g. [LTL06, WVO+09]. Hose et al. [HLS06] for example focused on Peer Data Management Systems and proposed an algorithm identifying relaxed Skyline results. Skyline queries have also been studied in other distributed environments, such as web information systems [BGZ04, LYLC06], mobile applications [XC10], or parallel shared-nothing architectures [VP10, VDK08]. Also distributed Skylines are relevant with respect to different data types, such as streamed [SHZ+10] or uncertain data [DJ10]. A detailed survey on the state-of-the-art in Skyline processing in distributed environments is given in [HV12].

All mentioned approaches from preference specification to distributed Skyline algorithms have in common that they are of academic nature and none of them has ever been integrated into a product release of a commercial database system. This paper describes the first commercial distributed Skyline environment for *Preference Analytics*.

3 Preference Analytics in EXASolution

Data analytics has become a key priority for many companies to enable a better decision-making process. *Preference Analytics* which integrates preference computations with data analytics enables users to define wishes and to extract the best objects from a given dataset. Preference Analytics is available in EXASolution via SKYLINE which seamlessly integrates with other analytical features. In this section we introduce the basics of SKYLINE; thereby commercializing the ideas in [Kie02, KEW11] by modifying the syntactical terms and introducing expressions and partitions in the preference clause.

3.1 Syntax

SKYLINE is available in EXASolution by an extension of the **SELECT**, **DELETE**, and **UPDATE** statements. In this paper, we only consider the **SELECT** statement because this is the most important use case for Skylines. Furthermore, using **DELETE** and **UPDATE** with Skylines is straightforward.

SELECT statements are extended by an *preferring clause* (Figure 3) which is introduced by the keyword **PREFERRING** (as in [KEW11]), and followed by a *preference term* (Figure 4) and an optional **PARTITION BY** sub-clause.

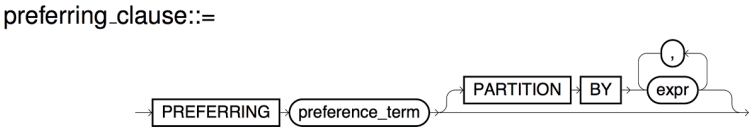


Figure 3: Preference clause in EXASolution.

The **PREFERRING** clause can be defined after the **WHERE** condition of a **SELECT** statement. It can contain the elements depicted in Figure 4 and is evaluated after the **WHERE** clause.

- **PARTITION BY**: If you specify this option, then the preferences are evaluated separately for each partition.
- **HIGH** and **LOW**: Defines whether an expression should have high or low values. Please note that any numerical expressions are expected here.
- *Boolean expressions*: In case of boolean expressions, the elements are preferred where the condition results in TRUE.
- **PLUS**: Via the keyword **PLUS**, multiple expressions of the same importance can be specified, i.e. the expressions form a Pareto preference query.
- **PRIOR TO**: With this clause you can nest two expressions hierarchically. The second term will only be considered if two elements have a similar value for the first term.
- **INVERSE**: By using the keyword **INVERSE**, you can create the dual preference expression. Hence, the expression **LOW** `price` is equivalent to **INVERSE**(**HIGH** `price`).

Note that in contrast to existing approaches as discussed in Section 2 it is possible to use arbitrary mathematical and boolean expressions in a preference term.

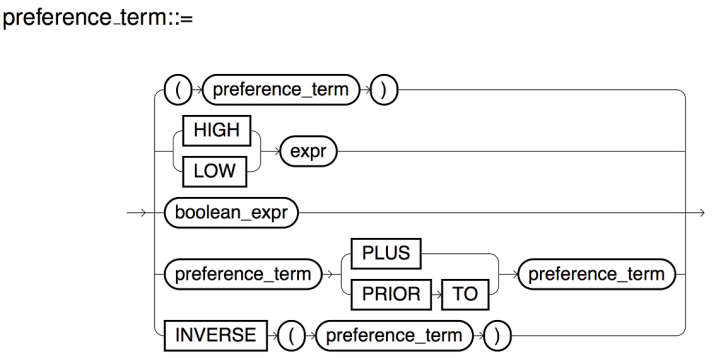


Figure 4: Preference term in EXASolution.

3.2 Evaluation of Preference Queries in EXASolution

EXASolution is a distributed and high-performance parallel database. It typically runs on a number of machines in a cluster. Data is distributed horizontally among all machines. This means that every machine contains parts of all (temporary or permanent) tables and result sets. When answering queries, an important goal of all algorithms in EXASolution is to maximize utilization of the available hardware.

Before we go into the details of the Skyline algorithm in EXASolution, we briefly describe the problem of Skyline computation. In general the Skyline contains those objects of a given dataset D that are not dominated by any others for a defined comparison function. In this paper we define the skyline as the set of best objects of a preference query.

More formally, the Skyline of a dataset D is defined by the maxima in D according to the ordering $<_P$, where $x <_P y$ for d-dimensional tuples $x = (x_1, ..., x_d)$, $y = (y_1, ..., y_d) \in D$ means *"y is better than x"* concerning the preference P. The *SKYline* set is

$$\text{SKY}(D) = \{t \in D | \nexists u \in D : t <_P u\}$$

The basic algorithm of SKYLINE in EXASolution belongs to the block-nested-loop class (BNL) [BKS01] of algorithms that filter the dataset using a compare function. The block-nested-loop algorithm linearly scans over the input dataset D. The idea of this algorithm is to continuously maintain a *window* (or block) of tuples in main memory containing the maximal elements with respect to the data read so far. When a tuple $p \in D$ is read from the input, p is compared to all tuples of the window and, based on this comparison, p is either eliminated, or placed into the window. Three cases can occur: First, p is dominated by a tuple within the window. In this case, p is eliminated and will not be considered in

future iterations. Of course, p need not be compared to all tuples of the window in this case. Second, p dominates one or more tuples in the window. In this case, these tuples are eliminated; that is, these tuples are removed from the window and will not be considered in future iterations while p is inserted into the window. And third, p is incomparable with all tuples in the window. In this case p is inserted into the window. At the end of the algorithm the window contains the maximal elements, i.e., *the Skyline*. The best case complexity is of the order $\mathcal{O}(n)$; n being the number of input tuples. In the worst case, the complexity is of the order of $\mathcal{O}(n^2)$.

The major advantage of a BNL-style algorithm is its simplicity and suitability for computing the maxima of arbitrary partial orders.

Given these constraints, the general design goals of the algorithm are straightforward:

- Optimize for interesting Skylines to be computed fast (and trade that for non-interesting ones tending to be slower)
- Compute Skyline in parallel and distributed on all machines

For the first goal we assume that smaller Skylines are more interesting than larger ones. This idea is based on the applications where Skyline is used to create short lists. The second goal is facilitated by the *additivity* [HV12] and the *distributability* of the Skyline operator. Additivity means that given n datasets D_i corresponding to n machines, the Skyline objects are the same if the Skyline operator is evaluated on (i) the union of the n datasets or (ii) first on each set separately and then once more on the union of the result set.

Property 1 (Additivity of the Skyline operator)**.** *Given a dataset D and n datasets D_i such that*

$$D = D_1 \cup \ldots \cup D_n$$

Then the following equation holds:

$$\mathit{SKY}(D_1 \cup \ldots \cup D_n) = \mathit{SKY}\left(\mathit{SKY}(D_1) \cup \ldots \cup \mathit{SKY}(D_n)\right)$$

Distributability means that members of the Skyline can be computed in the cluster with distributed data by testing them on each machine in the cluster independently. If they stand the test on all machines, they are part of the global Skyline. We formally specify this new outcome in the next property.

Property 2 (Distributability of the Skyline operator)**.** *Given a finite dataset D and n datasets D_i such that*

$$D = D_1 \cup \ldots \cup D_n$$

Then for each $d \in D$:

$$d \in \mathit{SKY}(D) \Leftrightarrow d \in (\mathit{SKY}(D_1 \cup \{d\}) \cap \ldots \cap \mathit{SKY}(D_n \cup \{d\}))$$

Proof. "⇒": Let $d \in \text{SKY}(D)$ and $d \notin (\text{SKY}(D_1 \cup \{d\}) \cap \ldots \cap \text{SKY}(D_n \cup \{d\}))$, then for some D_i, $d \notin \text{SKY}(D_i \cup \{d\})$. So there exists some element $d_0 \in D_i$ which dominates d. Since d_0 also is in D, $d \notin \text{SKY}(D)$ which contradicts the assumption.

"⇐": Let $d \in (\text{SKY}(D_1 \cup \{d\}) \cap \ldots \cap \text{SKY}(D_n \cup \{d\}))$ and $d \notin \text{SKY}(D)$. Then there exists $d_0 \in D$ which dominates d. Furthermore, there is some D_i with $d_0 \in D_i$ for which then $d \notin \text{SKY}(D_i \cup \{d\})$, which contradicts the assumption. □

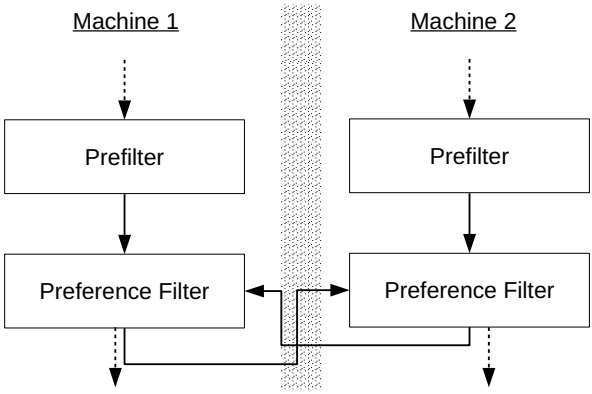


Figure 5: Skyline algorithm on a cluster with two machines

Based on Property 1 and 2, EXASOL's Skyline algorithm (see Figure 5) conceptually works in three steps:

1. Local pre-filtering
2. Local filtering
3. Global filtering

During *local pre-filtering*, subsets of the dataset are filtered independently on every machine fully in parallel, typically yielding a much smaller set of candidates per machine. Each of the datasets comprises a small Skyline but their union does not need to do so. This step corresponds to the inner $\text{SKY}(\cdot)$ operations according to Property 1.

After pre-filtering stopped on all machines. *Local* and *global filtering* are executed in parallel fashion, therby maximizing hardware utilization. Per node, candidates are compared to all other candidates (quadratic cost in the number of Skyline candidates). Candidates that turn out to be local solutions are sent off to the next machine in the cluster and compared to the local candidates there. Following this process, once a candidate reaches the machine it originated from it is known to be a global solution and written into the result set. Local filtering corresponds to the outer $\text{SKY}(\cdot)$ operations according to Property 1. The test on remote-machines corresponds to the right-hand side in Property 2.

4 Elaborate Examples

In this section we present elaborate examples which show the power of Preference Analytics and its advantage in comparison to standard SQL.

During this section we use the sample data in Table 1, which presents an ***energymap***, an extract of renewable energy sources in Germany. The full dataset is publicly available at *www.energymap.info*. The dataset contains information on solar stations, wind farms, biomass gas facilities, and hydro stations, in total more than 1.5M tuples.

generator_type	town	gpscoord	power_output	tso	...
Solarstrom	Weismain	POINT (50.05 11.233)	379865	TenneT TSO GmbH	
Solarstrom	Dingolfing	POINT (48.633 12.5)	53582	TenneT TSO GmbH	
Biomasse	Papenburg	POINT (53.067 7.4)	20000	TenneT TSO GmbH	
Biomasse	Emden	POINT (53.367 7.217)	19975	TenneT TSO GmbH	
Biomasse	Neumarkt	POINT (49.28 11.463)	19900	TenneT TSO GmbH	
Solarstrom	Fuchsstadt	POINT (50.1 9.933)	18737	TenneT TSO GmbH	
...	...	...	...	...	...

Table 1: Sample dataset of renewable energy sources in Germany (see *www.energymap.info*; please note that in order to take advantage of EXASolution's GEOMETRY type, the original columns *GPS-Lat* and *GPS-Lon* have been combined into the column *gpscoord* for our example).

Assume that one wants to choose the source of power for his company in Nuremberg. Given the `energymap` sample data in Table 1 the goal is to find suitable power stations. Since there is no power station exactly at the company's location, we have to consider the trade-off between big power stations further away and little power stations nearby.

Example 2 (Energymap – First attempt)**.** *The first attempt* without *Preference Analytics could be to find all interesting power stations using a* ***WHERE*** *clause to select only big power stations which are not too far away and very close power stations which are not too small, cf. Figure 6. We use the* ***ST_DISTANCE*** *function to compute the distance of each power station to the company's location coordinates* `N 49.47 E 11.041`*.*

```
SELECT (em.power_output/1000000) megawatts,
       ST_DISTANCE('POINT(49.47 11.041)', em.gpscoord) AS dist,
       em.generator_type, em.town, em.tso AS operator
FROM energymap em
WHERE em.gpscoord IS NOT NULL
-- the power station is nearby
-- (< 1.0 degree straight line distance, roughly 100km)
AND LOCAL.dist < 1.0
-- and it is relatively powerful (> 4 MW)
AND em.power_output > 4000000
ORDER BY em.power_output DESC;
```

Figure 6: Standard SQL query to find big power stations not too far away and very close power stations which are not too small.

One problem to notice here is that this query completely ignores middle sized power stations and those that are in the middle distance. Some of these might be a good solution. So this query is not satisfying and does not produce a usable short list of power stations. In fact, the query above returns no result at all which is an example of the empty result set effect (see Section 1) which is typical for such scenarios.

Example 3 (Energymap cont'd – Second attempt)**.** *The second attempt is to calculate a score based on a function of distance and power.*

```
-- We guess a suitable function of distance and power,
-- calculate a score for each station and consider the first 10 results
SELECT (em.power_output/1000000) megawatts,
       ST_DISTANCE('POINT(49.47 11.041)', em.gpscoord) AS dist,
       em.generator_type, em.town, em.tso AS operator
FROM energymap em
WHERE em.gpscoord IS NOT NULL
ORDER BY em.power_output/1000 + 100000/(1 + 10 * LOCAL.dist)
DESC LIMIT 10;
```

Figure 7: Calculate scores based on a function of distance and power.

However, there is no universal accepted formula for this task so we have to guess the function. If we are not satisfied with the results we have to change the function, and so forth. This is an iterative and time consuming approach. Furthermore, this process may lead to results which are further away and less powerful than other power stations. There is no protection from getting worse power stations over better ones. Furthermore, what about really powerful power stations further away? And what about really near small power stations?

Example 4 (Energymap cont'd – Preference Analytics)**.** *Finally using* SKYLINE *leads to a short list of Pareto optimal results. All we have to do is to add the* ***PREFERRING*** *clause for* low distance *and* high power output *to the query, cf. Figure 8.*

```
SELECT (em.power_output/1000000) megawatts,
       ST_DISTANCE('POINT(49.47 11.041)', em.gpscoord) AS dist,
       em.generator_type, em.town, em.tso AS operator
FROM energymap em WHERE em.gpscoord IS NOT NULL
PREFERRING LOW LOCAL.dist PLUS HIGH em.power_output
ORDER BY em.power_output DESC;
```

Figure 8: Use Preference Analytics to eliminate sub-optimal candidates.

The algorithm for Skyline computation in EXASolution is very efficient and runs distributed and parallel on all the machines of an EXASolution cluster. On a small test cluster of two rather dated servers, the result is computed in less than 1 second on the energymap dataset having more than 1.5M tuples. Using Preference Analytics one can compute the optimal trade-off of distance and power, creating a short list of highly interesting objects, ranging from a massive solar power station more than 400km away to tiny stations next door, cp. Table 2. From this set the user can make the final decision, thereby weighing the personal preferences for the best object.

megawatts	dist	generator_type	town	operator
1.468280	2.343107338557071	Solarstrom	Herrenberg	TransnetBW GmbH
0.626608	0.484542051838642	Solarstrom	Schirradorf	TenneT TSO GmbH
0.019900	0.462800172860813	Biomasse	Neumarkt	TenneT TSO GmbH
0.009200	0.374518357360488	Solarstrom	Pleinfeld	TenneT TSO GmbH
0.008213	0.277218325512582	Solarstrom	Heilsbronn	TenneT TSO GmbH
0.006125	0.093171884171138	Solarstrom	Zirndorf	TenneT TSO GmbH
0.006025	0.080993826925266	Biomasse	Sandreuth	TenneT TSO GmbH
0.001999	0.074060785845142	Biomasse	Fürth	TenneT TSO GmbH
0.000856	0.069570108523705	Solarstrom	Fürth	TenneT TSO GmbH
0.000451	0.034828149534534	Solarstrom	Nürnberg	TenneT TSO GmbH

Table 2: Result of the Preference Analytics query in Figure 8.

Example 5 (Energymap – Standard SQL)**.** *Since Skyline queries are not outside the expressive power of standard SQL [BKS01] we want to present the query in Example 4 in SQL using an inelegant* ***WHERE NOT EXISTS*** *clause. Doing so, this is cumbersome and not practical and leads to a much longer runtime of several minutes, especially for large datasets and many dimensions.*

```
SELECT (emo.power_output/1000000) megawatts,
       ST_DISTANCE('POINT(49.47 11.041)', emo.gpscoord) AS disto,
       emo.generator_type, emo.town, emo.tso AS operator
FROM energymap emo
WHERE emo.gpscoord IS NOT NULL
AND NOT EXISTS (
  SELECT (emi.power_output/1000000) megawatts,
         ST_DISTANCE('POINT(49.47 11.041)', emi.gpscoord) AS disti
  FROM energymap emi
  WHERE emi.gpscoord IS NOT NULL
  AND (
   -- no closer stations that are at least as powerful
   (LOCAL.disti < LOCAL.disto AND emi.power_output >= emo.power_output) OR
   -- no more powerful stations that are at least as close
   (LOCAL.disti <= LOCAL.disto AND emi.power_output > emo.power_putput)
  )
);
```

Figure 9: Preference Analytics query in standard SQL.

In our next example we would like to show that, as indicated in Figure 4, arbitrary expressions can be used in EXASolution's Preference Analytics.

Example 6. *If for some reason the user in Example 4 had an affinity to the location* Pleinfeld*, he could use a preference like in Figure 10. Here the standard SQL expression* **`LIKE`** *is used to find all towns which have the term* `PLEINFELD` *in their name. Then all towns are preferred where the condition results to* TRUE*.*

```
SELECT (em.power_output/1000000) megawatts,
       ST_DISTANCE('POINT(49.47 11.041)', em.gpscoord) AS dist,
       em.generator_type, em.town, em.tso AS operator
FROM energymap em
WHERE em.gpscoord IS NOT NULL
PREFERRING UPPER(em.town) LIKE '%PLEINFELD%'
           PLUS LOW LOCAL.dist PLUS HIGH em.power_output
ORDER BY em.power_output DESC;
```

Figure 10: Preference with Boolean expression.

5 Experiments

5.1 Test Environment

All experiments for Skyline processing have been performed on a powerful EXASolution cluster [KMZG13]. EXASolution is a parallelized relational DBMS which runs on a cluster of standard hardware servers. Following the *single program, multiple data* (SPMD) model, on each node the identical code is executed simultaneously. The data is stored in a column-oriented way and proprietary in-memory compression methods are used. Furthermore, EXASolution is designed to run in-memory, although data is persistently stored on disc following the ACID rules. EXASolution supports the SQL Standard 2003 and can be integrated via standard interfaces like ODBC, JDBC or ADO.NET. EXASolution runs on EXASOL's own Cluster Operating System (EXACluster OS). It is based on standard Linux and provides functionality for parallel programs. In our tests we used a cluster of 25 Dell PowerEdge R720xd servers which supports 1000 simultaneously executing threads.

5.2 Dataset

In many research papers on traditional Skylines the well-known real-world datasets NBA[2], Household[3], or Zillow[4] are used to examine the performance of preference query process-

[2] `www.nba.com`
[3] `www.ipums.org`
[4] `www.zillow.com`

ing [YSZ09, HV12, TBPY13, EK14]. The NBA dataset is a small 5-dimensional dataset containing 17264 tuples, where each entry records performance statistics for a NBA player. Following [CJT+06], NBA is fairly correlated and therefore is not a very challenging case for Skyline computation. Household is a larger 6-dimensional dataset having 127931 points. The Zillow dataset contains more than 2M entries about real estates in the United States. In a distributed environment synthetic datasets with less than 10M entries are often used to examine the performance of parallel Skyline processing. However, such a small input size is not really a challenge for a high-performance database management system such as EXASolution.

Currently there exists no real world benchmark for Skyline queries which supports large datasets. In order to gain access to realistic data sets with large volumes, we use the dataset which is used in the TPC-H benchmark[5] for analytical databases. The TPC-H benchmark is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

By using the TPC-H dataset to evaluate Skyline queries, we gain access to data at nearly arbitrary volumes. Data volumes in TPC-H are defined by *scale factors*, where a scale factor of 1 refers to 1 GB of data.

5.3 Queries

In this section we present queries for Preference Analytics based on queries in the TPC-H benchmark but expanded by **PREFERRING** clauses. We used queries having simple preferences (Q0) and complex queries using Pareto or Prioritization as well as partitioning (Q14). All queries are given below.

Q0:
```
SELECT p_partkey, p_size, ps_supplycost
FROM part, partsupp
WHERE p_partkey = ps_partkey
PREFERRING LOW ps_supplycost;
```

Q1:
```
SELECT p_partkey, p_size, ps_supplycost
FROM part, partsupp
WHERE p_partkey = ps_partkey
PREFERRING LOW ps_supplycost PRIOR TO HIGH p_size;
```

Q2:
```
SELECT p_partkey, p_size, ps_supplycost
FROM part, partsupp
WHERE p_partkey = ps_partkey
PREFERRING LOW ps_supplycost PLUS HIGH p_size;
```

[5] http://www.tpc.org/tpch/spec/tpch2.17.0.pdf

Q3:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING HIGH p_size PRIOR TO LOW s_acctbal;
```

Q4:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING HIGH p_size PLUS LOW s_acctbal;
```

Q5:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING (LOW ABS(ps_supplycost - 500) PLUS LOW ABS(s_acctbal))
           PRIOR TO LOW ABS(p_size - 12);
```

Q6:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING HIGH p_size PLUS LOW ps_supplycost PLUS LOW s_acctbal;
```

Q7:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal, l_orderkey
FROM part, partsupp, supplier, lineitem
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
        AND l_suppkey = s_suppkey AND l_partkey = p_partkey
PREFERRING (LOW ps_supplycost PLUS LOW s_acctbal PLUS HIGH p_size)
           PRIOR TO HIGH l_quantity;
```

Q8:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal,
       l_orderkey, l_quantity
FROM part, partsupp, supplier, lineitem
WHERE p_partkey = ps_partkey and ps_suppkey = s_suppkey
      AND l_suppkey = s_suppkey AND l_partkey = p_partkey
PREFERRING LOW ABS(ps_supplycost - 300) PLUS LOW ABS(s_acctbal)
   PLUS LOW CASE WHEN p_size <= 10 THEN 0
                 ELSE p_size - 10 END
   PLUS LOW CASE WHEN l_quantity >= 20 AND l_quantity <= 30 THEN 0
                 ELSE LEAST(ABS(l_quantity-20),
                            ABS(l_quantity-30)) END;
```

Q9:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal, l_orderkey
FROM part, partsupp, supplier, lineitem
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
      AND l_suppkey = s_suppkey AND l_partkey = p_partkey
PREFERRING LOW ps_supplycost PLUS LOW s_acctbal PLUS
           HIGH p_size PLUS HIGH l_quantity;
```

Q10:
```
SELECT ps_suppkey, p_partkey, p_size, ps_supplycost
FROM part, partsupp where p_partkey = ps_partkey
PREFERRING LOW ps_supplycost PRIOR TO HIGH p_size
   PARTITION BY ps_suppkey;
```

Q11:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING (LOW ABS(ps_supplycost - 500) PLUS LOW ABS(s_acctbal))
           PRIOR TO LOW ABS(p_size - 12)
    PARTITION BY s_suppkey;
```

Q12:
```
SELECT ps_suppkey, p_partkey, p_size, ps_supplycost, s_acctbal
FROM part, partsupp, supplier
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
PREFERRING HIGH p_size PLUS LOW s_acctbal
   PARTITION BY ps_suppkey;
```

Q13:
```
SELECT p_partkey, p_size, ps_supplycost, s_acctbal,
       l_orderkey, l_quantity
FROM part, partsupp, supplier, lineitem
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
      AND l_suppkey = s_suppkey AND l_partkey = p_partkey
PREFERRING LOW ABS(ps_supplycost - 300) PLUS LOW ABS(s_acctbal)
   PLUS LOW CASE WHEN p_size <= 10 THEN 0 ELSE p_size - 10 END
   PLUS LOW CASE WHEN l_quantity >= 20 AND l_quantity <= 30 THEN 0
            ELSE LEAST( ABS(l_quantity-20), ABS(l_quantity-30)) END
   PARTITION BY s_suppkey;
```

Q14:
```
SELECT ps_suppkey, p_partkey, p_size,
       ps_supplycost, s_acctbal, l_orderkey
FROM part, partsupp, supplier, lineitem
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
      AND l_suppkey = s_suppkey AND l_partkey = p_partkey
PREFERRING LOW ps_supplycost PLUS LOW s_acctbal
           PLUS HIGH p_size PLUS HIGH l_quantity
    PARTITION BY ps_suppkey;
```

5.4 Results

In Tables 3 - 6 we used the TPC-H dataset with a scaling factor of 1 to 1000, i.e. up to 1000 GB of data. The column titled *join size (rows)* contains the number of rows after the join in the SQL query. The *result size (rows)* denotes the Skyline size, i.e. the number of objects after the preference selection. The last column *elapsed time* represents the runtime of the query in seconds.

Since there is no other commercial implementation of Skyline and other approaches for Preference Analytics (cp. Section 2) are of academic nature, this is the first benchmark of Skylines on large datasets.

Query	join size (rows)	result size (rows)	elapsed time (seconds)
Q0	800 000	6	0.094
Q1	800 000	1	0.049
Q2	800 000	1	0.049
Q3	800 000	3	0.058
Q4	800 000	3	0.059
Q5	800 000	12	0.066
Q6	800 000	33	0.077
Q7	6 001 215	35	0.184
Q8	6 001 215	51	0.197
Q9	6 001 215	73	0.206
Q10	800 000	10 000	1.109
Q11	800 000	10 001	0.695
Q12	800 000	19 646	0.886
Q13	6 001 215	58 119	3.098
Q14	6 001 215	101 997	3.610

Table 3: Benchmark results for TPC-H scale factor 1.

Query	join size (rows)	result size (rows)	elapsed time (seconds)
Q0	8 000 000	91	0.077
Q1	8 000 000	5	0.053
Q2	8 000 000	5	0.056
Q3	8 000 000	1	0.086
Q4	8 000 000	4	0.086
Q5	8 000 000	7	0.093
Q6	8 000 000	53	0.110
Q7	59 986 052	56	0.445
Q8	59 986 052	66	0.405
Q9	59 986 052	140	0.483
Q10	8 000 000	100 001	6.236
Q11	8 000 000	100 001	5.622
Q12	8 000 000	199 263	7.752
Q13	59 986 052	580 554	34.033
Q14	59 986 052	1 000 383	38.765

Table 4: Benchmark results for TPC-H scale factor 10.

Query	join size (rows)	result size (rows)	elapsed time (seconds)
Q0	80 000 000	816	0.216
Q1	80 000 000	20	0.085
Q2	80 000 000	20	0.094
Q3	80 000 000	4	0.364
Q4	80 000 000	4	0.618
Q5	80 000 000	10	0.357
Q6	80 000 000	48	0.377
Q7	600 037 902	49	4.595
Q8	600 037 902	41	5.746
Q9	600 037 902	125	4.856
Q10	80 000 000	1 000 007	55.782
Q11	80 000 000	1 000 024	61.296
Q12	80 000 000	2 000 115	79.933
Q13	600 037 902	5 496 641	350.081
Q14	600 037 902	9 268 943	406.202

Table 5: Benchmark result for TPC-H scale factor 100.

Query	join size (rows)	result size (rows)	elapsed time (seconds)
Q0	800 000 000	8029	1.178
Q1	800 000 000	138	0.342
Q2	800 000 000	138	0.349
Q3	800 000 000	15	2.785
Q4	800 000 000	15	2.618
Q5	800 000 000	2	2.627
Q6	800 000 000	34	2.755
Q7	5 999 989 709	104	52.394
Q8	5 999 989 709	64	57.664
Q9	5 999 989 709	346	64.447
Q10	800 000 000	10 000 079	649.118
Q11	800 000 000	10 000 225	719.784
Q12	800 000 000	20 020 772	924.670
Q13	5 999 989 709	141 220 077	6 899.464
Q14	5 999 989 709	243 220 024	9 620.591

Table 6: Benchmark result for TPC-H scale factor 1000.

Figure 11 presents the runtime for all queries in logarithmic scale for scale factors 1, 10, 100, and 1000 on the TPC-H benchmark dataset.

By examining the results, there is a clear tendency for queries with smaller result sizes to run fast and for queries with larger result sizes (at the same input size) to be slower. A tendency which corresponds to the trade-off described in Section 3.2.

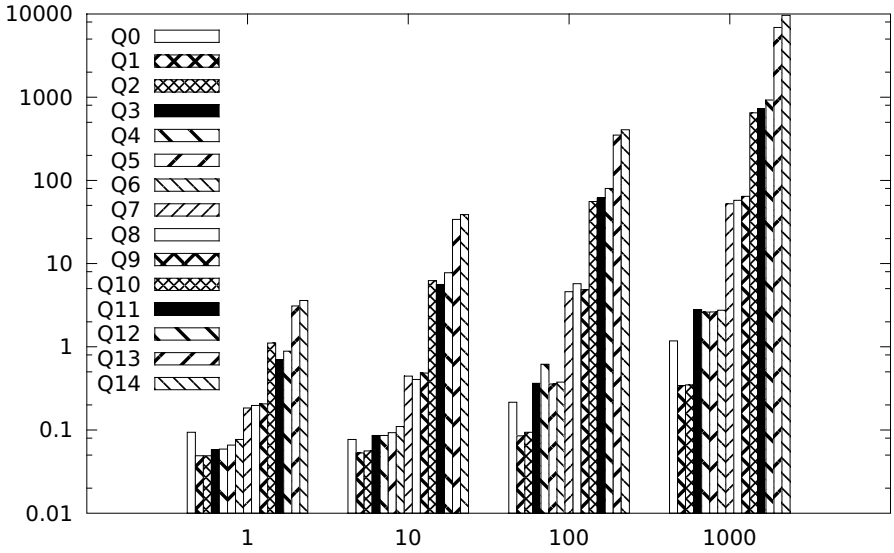


Figure 11: Runtime for Queries 0 to 14 in seconds (logarithmic scale) for scale factors 1, 10, 100, and 1000 on the TPC-H dataset.

6 Conclusion

During the last decade, data are increasingly stored in a distributed way, hence distributed query processing has become an important and challenging problem. On the other hand, the popularity of preference database queries due to their ability to identify sets of highly interesting objects in large databases makes it necessary to unify these two approaches. To tackle the problem of high performance Skyline processing in a distributed environment we integrated preference queries into EXASolution.

This paper provides insight into how Preference Analytics works in EXASolution, a high performance parallel and distributed engine for data analytics and data warehousing. It is the first (and by the time of writing also the only) commercial database that features Skylines natively and at scale. We present novel and expressive operations for preference construction and describe a distributed Skyline algorithm. Furthermore, a quantitative performance analysis is provided. Providing Skylines as an in-database analytical tool enables *Preference Analytics*, which facilitates focusing the results of advanced analytics on best-matching objects only, thereby enabling better and more agile decision-making.

References

[AK12] A. Arvanitis and G. Koutrika. Towards Preference-Aware Relational Databases. In *Proceedings of ICDE '12*, Washington, DC, USA, April 2012.

[BGZ04] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT '04: Proceedings of the 9th International Conference on Extending Database Technology*, volume 2992 of *LNCS*, pages 256–273, 2004.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[CCM13] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline Queries, Front and Back. *SIGMOD Rec.*, 42(3):6–18, October 2013.

[CDK06] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust Cardinality and Cost Estimation for Skyline Operator. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 64, Washington, DC, USA, 2006. IEEE Computer Society.

[CJT+06] C. Y. Chang, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On High Dimensional Skylines. In *EDBT '06: Proceedings of the 10th International Conference on Extending Database Technology*, volume 3896 of *LNCS*, pages 478–495, 2006.

[DJ10] X. Ding and H. Jin. Efficient and Progressive Algorithms for Distributed Skyline Queries over Uncertain Data. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 149–158. IEEE, 2010.

[EF09] H. Eder and W. Fang. Evaluation of Skyline Algorithms in PostgreSQL. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 334–337, New York, NY, USA, 2009. ACM.

[EK14] M. Endres and W. Kießling. High Parallel Skyline Computation over Low-Cardinality Domains. In *ADBIS '14: Advances in Databases and Information Systems*, pages 97–111. Springer, 2014.

[End14] M. Endres. A Survey on Selectivity Estimation for Preference Database Queries. In T. Robal H.-M. Haav, A. Kalja, editor, *Frontiers in Artificial Intelligence and Applications (FAIA)*, volume 270, pages 159–172. IOS Press, 2014.

[GRB11] M. Golfarelli, S. Rizzi, and P. Biondi. myOLAP: An Approach to Express and Evaluate OLAP Preferences. *TKDE*, 23(7):1050–1064, 2011.

[HLS06] K. Hose, C. Lemke, and K.-U. Sattler. Processing Relaxed Skylines in PDMS Using Distributed Data Summaries. In *Proceedings of CIKM '06*, pages 425–434, New York, NY, USA, 2006. ACM.

[HV12] K. Hose and A. Vlachou. A Survey of Skyline Processing in Highly Distributed Environments. *The VLDB Journal*, 21(3):359–384, June 2012.

[KEW11] W. Kießling, M. Endres, and F. Wenzel. The Preference SQL System - An Overview. *Bulletin of the Technical Commitee on Data Engineering, IEEE*, 34(2):11–18, 2011.

[Kie02] W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of VLDB '02*, pages 311–322, Hong Kong, China, 2002. VLDB Endowment.

[KMZG13] S. Klenk, S. Mandl, S. Zentgraf, and M. Golombek. Advanced Analytics mit der analytischen In-Memory Datenbank EXASolution. In *GI-Jahrestagung*, 2013.

[LEMK13] J. J. Levandoski, A. Eldawy, M. F. Mokbel, and M. E. Khalefa. Flexible and Extensible Preference Evaluation in Database Systems. *ACM Trans. Database Syst.*, 38(3):17:1–17:43, September 2013.

[LTL06] H. Li, Q. Tan, and W.-C. Lee. Efficient Progressive Processing of Skyline Queries in Peer-to-Peer Systems. In *Proceedings of the 1st International Conference on Scalable Information Systems*, pages 26–es, New York, NY USA, May 2006. ACM.

[LVDN14] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg. APSkyline: Improved Skyline Computation for Multicore Architectures. In *Proc. of DASFAA '14*, LNCS, 2014.

[LYLC06] E. Lo, K. Y. Yip, K.-I. Lin, and D. W. Cheung. Progressive Skylining Over Web-accessible Databases. *IEEE TKDE*, 57(2):122–147, May 2006.

[PKP+09] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel Skyline Computation on Multicore Architectures. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 760–771, Washington, DC, USA, 2009. IEEE.

[RJVDN09] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. AGiDS: A Grid-Based Strategy for Distributed Skyline Query Processing. In *Globe '09: Proceedings of the 2nd International Conference on Data Management in Grid and Peer-to-Peer Systems*, Globe '09, pages 12–23, Berlin, Heidelberg, 2009. Springer-Verlag.

[SHZ+10] S. Sun, Z. Huang, H. Zhong, D. Dai, H. Liu, and J. Li. Efficient Monitoring of Skyline Queries over Distributed Data Streams. *Knowl. Inf. Syst.*, 25(3):575–606, 2010.

[SKP11] K. Stefanidis, G. Koutrika, and E. Pitoura. A Survey on Representation, Composition and Application of Preferences in Database Systems. *ACM TODS*, 36(4), 2011.

[SLB10] J. Selke, C. Lofi, and W.-T. Balke. Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval. In *DASFAA '10: 15th International Conference on Database Systems for Advanced Applications*, volume 5982 of *LNCS*, pages 246–260, Tsukuba, Japan, 04/2010 2010. Springer.

[TBPY13] G. Trimponias, I. Bartolini, D. Papadias, and Y. Yang. Skyline Processing on Distributed Vertical Decompositions. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):850–862, 2013.

[VDK08] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based Space Partitioning for Efficient Parallel Skyline Computation. In *SIGMOD '08: Proceedings of the 34th SIGMOD International Conference on Management of data*, SIGMOD '08, pages 227–238, New York, NY, USA, 2008. ACM.

[VP10] G Valkanas and A N Papadopoulos. Efficient and Adaptive Distributed Skyline Computation. *Lecture Notes in Computer Science*, pages 1–18, April 2010.

[WVO+09] S. Wang, Q. H. Vu, B. C. Ooi, A. K. H. Tung, and L. Xu. Skyframe: A Framework for Skyline Query Processing in Peer-to-Peer Systems. *VLDB J.*, 18(1):345–362, 2009.

[WZF+06] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *EDBT '06: Proceedings of the 10th International Conference on Extending Database Technology*, volume 3896 of *LNCS*, pages 112–130. Springer, 2006.

[XC10] Y.-Y. Xiao and Y.-G. Chen. Efficient Distributed Skyline Queries for Mobile Applications. *Journal of Computer Science and Technology*, 25(3):523–536, 2010.

[YSZ09] L. Zhu Y., S., and Zhou. Distributed Skyline Retrieval with Low Bandwith Consumption. *TKDE*, 2009.

Generic Business Simulation Using an In-Memory Column Store

Lars Butzmann[1], Stefan Klauck[1], Stephan Müller[1], Matthias Uflacker[1], Werner Sinzig[2], and Hasso Plattner[1]

[1]Hasso Plattner Institute, University of Potsdam, Germany, {firstname.lastname}@hpi.de
[2]SAP SE, Walldorf, Germany, werner.sinzig@sap.com

Abstract:
Value driver trees are a well-known methodology to model dependencies such as the definition of key performance indicators. While the models have well-known semantics, they lack the right tool support for business simulations, because a flexible implementation that supports multidimensional, hierarchical value driver trees and data bindings is very complex and computationally challenging. This paper tackles this problem by proposing an approach for generic enterprise simulations which are based on value driver trees. Our approach is two-fold: we present the definition of a simulation meta model at design time, and the run-time simulation tool. The simulation meta model describes the structure of the dependency graph, the data binding, and the parametrization of the model to simulate data changes. The simulation tool can then be used to create and edit simulation model instances and run simulations in real-time by leveraging an in-memory column store. Besides the formal description of the approach, this work presents a prototypical implementation of the simulation tool and an evaluation using data of a consumer packaged goods company.

1 Introduction

Companies invest a significant amount of time in their yearly budgeting process and the related quarterly or monthly forecasts. Especially in the higher management levels, this process is often questioned in terms of efficiency and effectiveness. Based on this deficiency, radical approaches such as *Beyond Budgeting* or alternative solutions such as *Balanced Scorecards* have been proposed. In this context, *Predictive Analytics* has been established with the goal to model cause and effect in an enterprise. This functionality can be used in the pre-budgeting process, be part of a forecast to evaluate scenarios in terms of their goal fulfillment, or try to support decisions in day to day operations.

Within the group of predictive analytics, value driver trees such as the DuPont model [CS00] are a well-known methodology. Examples for the usage of value driver trees can be found in several areas of an enterprise.

- **Procurement**: product costs depending on commodity prices, raw material purchase prices and currency exchange rates.

- **Production**: product costs depending on used energy, energy prices and production loss.
- **Sales**: sales depending on sales prices, behavior of competing companies, and available income of customers.
- **Finance**: profitability based on sales volume by market segmentation, sales prices, and currency exchange rates. Payment depending on delivery quality and adherence to delivery dates.

All these example use cases rely on linear relationships, which can be expressed by a single linear equation or by a set of linear equations with dependencies among another, as introduced by Zwicker [Zwi03]. The modeling of business processes using value driver trees provides a number of benefits: The decision making process can be supported by focusing on the key factors. This is accompanied by a reduction of the planning effort by focusing on the essentials. Also, it supports the collaboration among different business areas to create a common strategy. And if the key performance indicators are directly connected to the operational value drivers, planning for finance and accounting can become more realistic and efficient.

The challenge of using value driver trees is not the correct modeling of a company or the mathematical complexity of such a solution. It is more the acquisition of the required data, the limited capabilities of existing solutions and the expensive calculation of hierarchical dependencies on large data sets. The data is often not transparently available in ERP systems (e.g. bill of materials and routing coefficients). Also, it may be distributed among multiple systems (e.g. sales, production, and finance) and only be known implicitly (e.g. elasticity factors for the inverse demand function for products). Available solutions on the other hand are developed for specific use cases such as the calculation of product costs, the calculation of advertising effects, or the calculation of currency effects. With changing requirements however, the solution cannot be adapted but has to be reimplemented. One of the biggest drawbacks of current solutions is the run-time of calculations. While subsecond response times are required for a usage in the on-line decision making and planning process, a majority of existing solutions is running batch processes and therefore cannot meet this goal. Batch processes typically run several hours, most of the time over night.

A possible solution for this problem is the in-memory database technology. Previous work by Plattner [Pla09] has shown the benefits of columnar storage using main memory technology for analytical workloads. With the improved query performance, it is now possible to simplify complex data models [Pla14] that relied on materialized aggregates and other workarounds that were used to provide reasonable response times. The result is an increasing data transparency. Another benefit is the flexibility to query the data, even when the data sets are large and relations need to be included. This flexibility is important for retrieving the value drivers (sales volume, purchasing volume, expenses) as well as the coefficients between the drivers.

In this work, we introduce a solution to address the mentioned challenges and problems. First, we introduce a meta model how dependency graphs can be created and configured. Second, we explain the simulation tool and how it creates and runs simulations. Third, we

present a prototypical implementation of the HPI Business Simulator tool and demonstrate it using a real-world example. Finally, we analyze the query performance for both storage types, a row store and a column store.

2 Approach

Our approach is divided into three parts to build a tool for business simulations: the definition of a generic calculation model, the data binding and the simulation.

2.1 Calculation Model

Simulation models are hypergraphs. Each node has a name, available dimensions and can be connected with other nodes by *operations*, which are hyperedges. The simulation model specifies the available dimensions with their hierarchy levels. Time can for example be hierarchically structured into years, months, and days. The dimensions of a specific node are determined by the data binding or the operation and dimension specification of connected nodes in case the node has no data source.

Operations define the dependencies between nodes. Each operation has to define the way it calculates the values for the result nodes. Thereby, the approach can define the calculations of common operations as addition, subtraction, multiplication and division, which work in a similar way as for one-dimensional data. Combining the values of nodes with different levels of hierarchies may require a preaggregation phase. In other cases, the operation cannot be executed at all. The signature and algorithm of each operation has to be defined. The addition operation connects two nodes with the same dimensions and creates a node with values for these specific dimensions. The hierarchy level of the resulting node is the minimum of both input nodes.

2.2 Data Binding

Nodes of the calculation model can obtain their values in two ways. On the one hand, they can query their data directly from database tables. On the other hand, they can calculate their values by solving the equation defined by the operation between connected nodes and itself. For the first case, data bindings are required. Our work focuses on relational databases as a data source, although it is also possible to use other sources. Data bindings define the database connection and SQL queries for all supported dimension, filter criteria and hierarchy levels. A generic way to implement it without specifying each single possible query is to define the SQL query to request the data for all dimensions at the lowest hierarchy levels. Based on this data, filter criteria and aggregations can be applied to support selections and views of the data at higher hierarchy levels. Figure 1 shows a dependency graph with marked nodes for data sources.

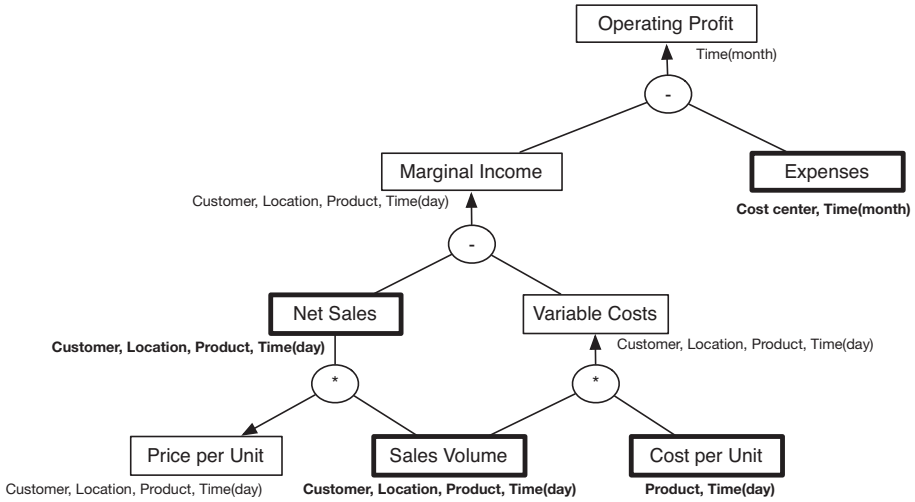


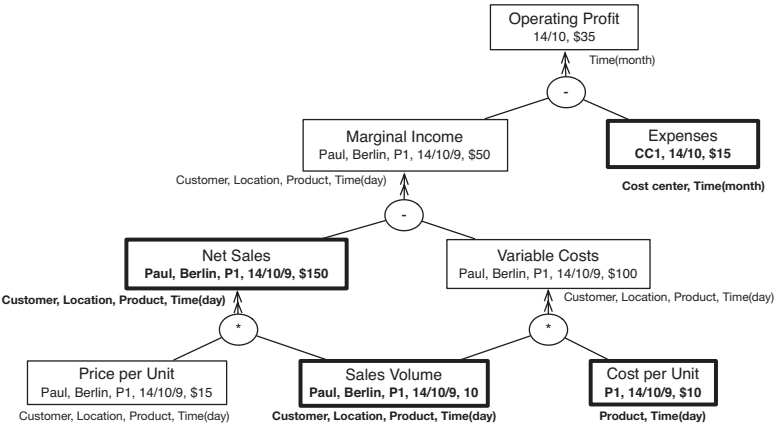
Figure 1: Dependency graph for the operation profit: Bold nodes represent data sources with their dimensions. The values of nodes which are marked by arrows can be calculated recursively by following the operations from which the arrows come from.

2.3 Simulation

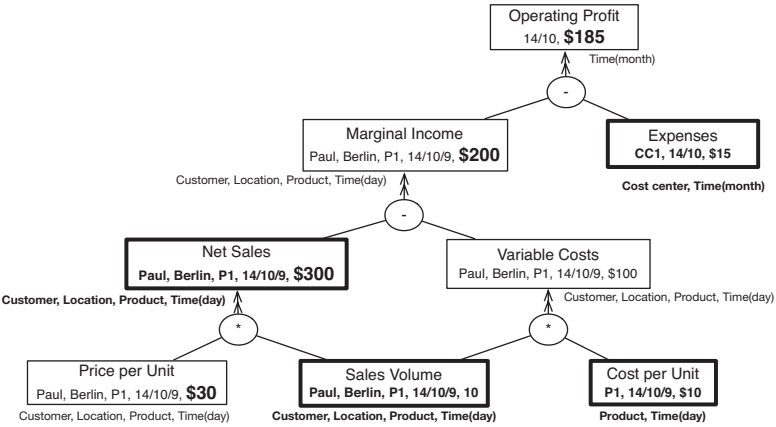
The simulation tool supports creating, changing and running simulation models. In general, different user groups are responsible for model editing and specifying simulation scenarios. The choice of KPIs and its adaption to better fit the company's structure is executed at management level. As the management level does not have a detailed knowledge of the data schema, it will delegate the task of defining data bindings to more technical staff. The individual simulations can then be done in collaboration with controllers. Hence, the tool has two modes to separate these activities: one for editing simulation model instances, the other for executing simulations. The editor mode comprises an interface to create dependency models as described in Section 2.1 and allows to specify the data binding. A graphical modeling tool can be used as user-friendly interface, as well as a text editor to describe the model with structured text, e.g. JSON or XML.

Simulation model instances could then be validated before switching to the simulation mode. The simulation mode allows drilldowns, i.e. filtering the node values by dimensions and hierarchy levels. The number of dimensions specifies the number of drilldown possibilities. The depth of the hierarchy specifies how deep the drilldown goes. If a node does not support the current drilldown level, it is excluded from the simulation. For each drilldown level, simulation parameters can be specified. Thereby, the model has to define directions for edges to specify which nodes are affected by the parameterization of connected nodes.

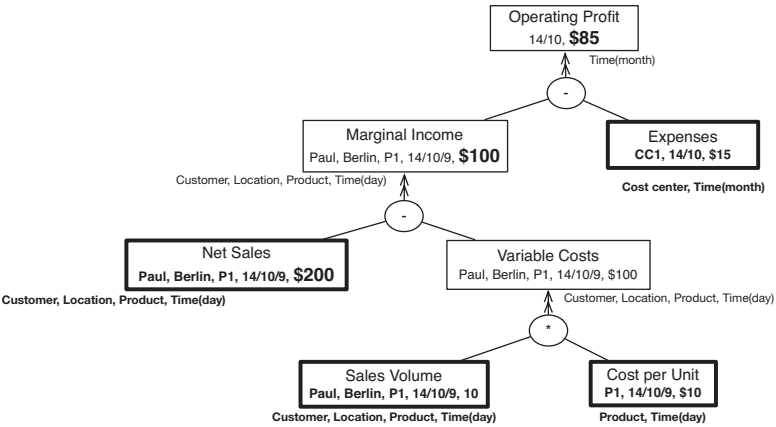
Figure 2 shows an example of a simulation. Based on the dependency graph as given in



(a) Initial state with data from the database table.



(b) Simulation with an absolute value for price per unit.



(c) Simulation with an absolute value for net sales.

Figure 2: Examples of Simulations.

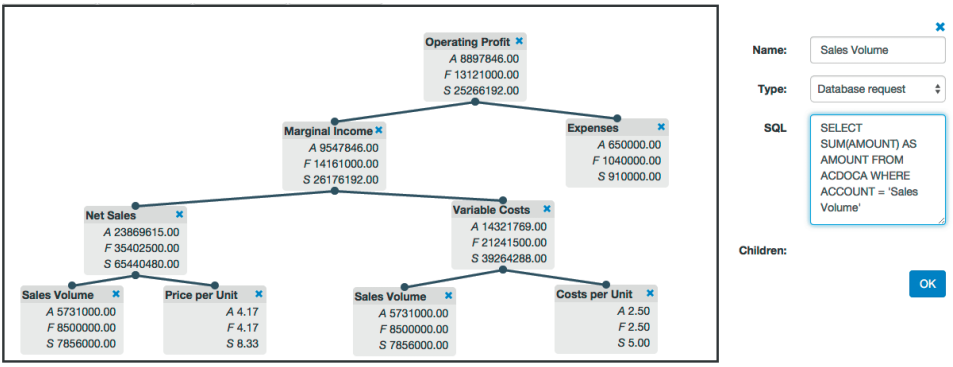


Figure 3: The editor mode of the HPI Business Simulator.

Figure 1 with data sources for net sales, sales volume, cost per unit and expenses, the other nodes can be calculated forming the graph as presented in Figure 2a. The values are the basis of the simulation. The double arrows specify in which directions the values are propagated. Each value, independent whether it was directly queried from a database table or was calculated, can be overwritten by simulation parameters. Figure 2b shows the simulation of an increased price per unit ($30 instead of $15) and how it propagates via the endpoints with double arrows. When parameterizing nodes with incoming double arrows, e.g. setting net sales to $200, the connected nodes are ignored (cf. Figure 2c).

3 Proof of Concept

In this section, we explain the capabilities of the HPI Business Simulator. For a proof of concept (POC), we have engaged with a Fortune 500 company in the consumer packaged goods (CPG) industry and discussed their deficiencies and needs in the area of business simulation. Based on the input and data set they provided to us, we created a prototype of our simulation model.

3.1 HPI Business Simulator

In the first step, we created an instance of a simulation model using the editor mode of our tool. The created graph consists of 8 nodes (the node for sales volume is represented twice) and 4 multidimensional attributes called dimensions. The complete tree is shown in Figure 3.

```
{
    "dimensions": {
        "time": ["year", "month", "day"],
        "product": ["product"],
        "customer: ["customer"],
        "region": ["country", "city"]
```

```
    },
    "nodes": {
        "node1": {
            "name": "Sales Volume",
            "dimensions": {"customer": "customer", "region": "country", "time": "day
                ", "product": "product"},
            "unit": "kg",
            "SQL": "SELECT SUM(AMOUNT) AS AMOUNT FROM ACDOCA WHERE ACCOUNT = 'Sales
                Volume'"
        },
        "node2": {
            "name": "Price per Unit",
            "dimensions": {"customer": "customer", "region": "country", "time": "day
                ", "product": "product"},
            "unit": "USD/kg",
        },
        "node3": {
            "name": "Net Sales",
            "dimensions": {"customer": "customer", "region": "country", "time": "day
                ", "product": "product"},
            "unit": "USD",
            "SQL": "SELECT SUM(AMOUNT) AS AMOUNT FROM ACDOCA WHERE ACCOUNT = 'Net
                Sales'"
        }
    },
    "operations": {
        "node3": "node1 * node2"
    }
}
```

Listing 1: An extract of the model definition using the JSON format.

We use the JavaScript Object Notation (JSON) as a data format to store the model information. Each node is represented as an object, specifying its name and corresponding properties. The ID is used as key for the corresponding node object. Listing 1 is an example for a subgraph that appears in Figure 3. The first level of the JSON structure contains three values, the existing dimensions, the nodes itself, and the operations which represent the connections between the nodes. As explained in Section 2.1, a node consists of a set of properties. The node with the name *Sales Volume* has a data source and therefore has an SQL query that defines how its data can be retrieved. In contrast, the connected node *Price per Unit* has no data binding and calculates its values using the formula defined in the operations list. In this subgraph, the units are not equal for all nodes. In cases where the units are not the same, the editor ensures that the operations return the correct units. While creating the model, the editor constantly checks the model for consistency and correctness. In this example, the unit of *Sales Volume* is kg and the unit of *Price per Unit* is USD/kg. Since the unit *kg* is in the denominator and numerator, they cancel each other out, leaving USD as the unit for *Net Sales*.

3.1.1 Consumer Packaged Goods Data

The underlying data set we got had a normalized data schema with one transactional table (ACDOCA) and multiple corresponding dimension tables. The transactional table consists of approximately 300 million line items which represent a period of 5 years. Dimensions that were included are the account type, cost center, product information, and location information. For our scenario, we used a denormalized data schema to remove the join

Table 1: Example data for the underlying ACDOCA table with four dimensions.

DocID	G/L Account	Cost Center	Product	Customer	Location	Date	Qty	Amount
01	COGS		Product 1	Paul	Berlin	10/9/2014	10	100.00
02	COGS		Product 2	Paul	Berlin	10/9/2014	5	75.00
03	Inventory					1/1/2014		-175.00
04	Revenue		Product 1	Paul	Berlin	10/9/2014	10	150.00
05	Revenue		Product 2	Paul	Berlin	10/9/2014	5	125.00
06	Receivables			Paul	Berlin	10/9/2014		275.00
07	Expenses	Cost Center 1				10/2014		15.00

complexity for the model creation process. Table 1 shows example data for the relevant columns used by the business simulator. The two main columns are *G/L Account* and *Quantity* respectively *Amount* which are included in every query. *G/L Account* describes the type of account a line item corresponds to. The other five dimension columns are only included into the SQL query when they are required.

Based on the customer feedback, the HPI Business Simulator (cf. Figure 3) shows three values per node. First, the actual value (*A*) for the current period which includes all values that exist until today. Second, the forecasted value (*F*) for the complete period that is based on A and plan data (*P*). Third, the simulated value (*S*) which is based on *F* and changes calculated by the simulation model. By default, the forecast and the simulation value are equal because no parameter has been changed. By adjusting a node with an absolute or relative value, the valued driver tree is recalculated and updated with the new results. As a result, the simulation value changes and consequently reflects the changes compared to the forecast value. The user is able to select certain filters to drill down into a dimension, e.g. a certain product. In this case, the adjustments are only applied for the selected dimensions.

3.2 In-Memory Column Store

A key enabler for our tool is in-memory column stores, such as SAP HANA [FCP+11]. By keeping all data in main memory, the run-time of analytical queries, as required for the business simulator, can be reduced significantly compared to traditional disk-based systems. The second benefit is the columnar data layout. Since the mapping of the simulation model only accesses columns that are required for a calculation, the amount of processed data is kept to a minimum. This functionality is especially beneficial in the context of enterprise systems where examples show that tables can have more than 300 columns. The columnar layout is also well suited for analytical queries, e.g. aggregate queries, where larger data sets of a column have to be scanned and aggregated.

A third benefit is an aggregate cache which is a transparent caching engine inside the database [MP13]. Other than traditional materialized views, the aggregate cache does not create a hard copy of the data and as a result does not return stale data. The aggregate cache leverages the internal table representation in certain column stores like SAP HANA or Hyrise [GKP+10] and always works on the latest data. During a typical simulation session, different scenarios are analyzed and compared. A scenario includes a set of changes in the value driver tree. The differences between the scenarios vary, but are usually small.

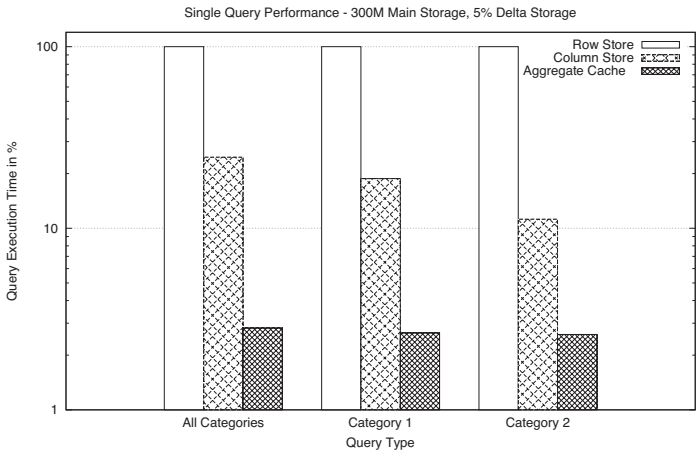


Figure 4: A performance comparison between a row store, a column store and a column store with an aggregate cache for three query types.

Consequently, the executed queries are similar and can therefore benefit from the aggregate cache. Further, the created scenarios are constantly reviewed and the updated forecast is compared to the simulated value. In that process, the simulator can rely on the aggregate cache to speed up the query execution and reduce the overall system load.

3.2.1 Performance Analysis

In our evaluation, we measured the performance of the queries using a row store, a column store and a column store with an aggregate cache inside SAP HANA [FCP+11]. The benchmarks have been conducted on a server featuring 128 CPUs (Intel Xeon X7560) with 2.27GHz and 24MB cache each. The entire machine was comprised of 512GB of main memory. The data set size on disk was approximately 130GB. Every benchmark was run at least five times and the displayed results are the median of all runs.

In Figure 4, the query performance for three types of queries is evaluated. The three types differ in the amount of data they have to aggregate. To demonstrate the impact of a column store and the aggregate cache, we compared the performance with an in-memory row store inside SAP HANA. The results are shown relatively to the row store, which serves as the baseline for the other two approaches. With the first query, the value of a single value driver, in this case *Net Sales*, for all categories was retrieved. Using the column store, the query is accelerated by a factor 4. For the aggregate cache, it is accelerated by a factor of 35. For the second and third query, the same query was used with a filter on the category. Category 1 is the biggest category including approximately 40% of all products. Category 2 is a smaller category including only 10% of the products. The execution time for the row store resulted in the same execution time as for the first query, even though the number of records that had to be aggregated was smaller. In contrast, the performance of the column store increased with an increasing selectivity resulting in an acceleration

of a factor 10 for the third query. The performance using the aggregate cache was only slightly faster for the second and third query (factor 38 compared to the row store). This is a result of its characteristic since it only caches the results of the main storage and still has to aggregate the delta storage on the fly. The records in the delta storage just slightly decreased for query 2 and 3. A deeper evaluation of the aggregate cache was done by Müller and Plattner [MP13].

The benchmark shows that a column store is superior to a row store for typical business simulator queries. However, more investigation and benchmarking is necessary, especially in the context of executing multiple similar queries at the same time, as required by the business simulator.

4 Related Work

Besides sophisticated controlling software, spreadsheets, MS Excel in particular, are the main simulation tools in enterprises. The RS-Controlling-System [1] offers Excel templates for target-performance comparisons of balance sheets, profit and loss statements and capital flows. Spreadsheets have an easy to understand interface, but do not build on consolidated enterprise data, which is stored in a RDBMS. On the other side, SQL lacks the support for array-like calculations as Witkowski et al. claim in [WBB+03]. Their idea is to combine both and offer a spreadsheet-like computation in RDBMS through SQL extensions. In [WBB+05], they continue that research and introduce a way to translate MS Excel computations in SQL. The simulation model is thereby expressed in MS Excel. This approach comes with two drawbacks. First, it does not encapsulate the definition of the simulation model so that it is not tangible, but only expressed by multiple formulas spread over many table cells. Second, MS Excel is bound to the two-dimensional representation and cannot visualize graphical dependencies very well.

To describe simulation models, Golfarelli et al. introduce a methodology to formally express and build what-if analyses in [GRP06]. They divide the process to design simulations into seven phases: goal analysis, business modeling, data source analysis, multidimensional modeling, simulation modeling, data design and implementation, and validation. To describe the actual simulation model they propose an extension of UML 2 activity diagrams [GR08]. However, they do not focus on how to develop generic simulation applications that are based on the formal descriptions.

5 Conclusions and Future Work

This paper proposes a new approach to build business simulations. It consists of a multidimensional generic model to describe the dependencies of value drivers as basis of simulations and the concept of a simulation tool to create, edit and parameterize model instances

[1] http://www.controllingportal.de/Shop/Excel-Tools/RS-Controlling-System.html

to simulate scenarios.

The main benefits of our solution can be summarized as follows: The generic business simulation approach enhances the focus on key factors that drive the business. Further, it reduces the planning effort and drives cross-functional collaboration and alignment. By directly using transactional data as a basis for the simulation, we exclude consistency issues. Most importantly, the simulation solution provides leadership with visibility into different alternatives to achieve desired outcomes.

The calculation model and database binding are described in a semi-formal way in prose text and JSON. Future work can formalize the model and the multidimensional operations. Of special interest are the possibilities and specifications of simulation changes. Beside, an investigation of optimization strategies for the required database queries, which base on groupings with different granularities, can be done.

References

[CS00] A.D. Chandler and S. Salsbury. *Pierre S. Du Pont and the Making of the Modern Corporation*. BeardBooks, 2000.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. In *ACM SIGMOD*, pages 45–51, 2011.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[GR08] Matteo Golfarelli and Stefano Rizzi. UML-Based Modeling for What-If Analysis. In *DaWak*, pages 1–12, 2008.

[GRP06] Matteo Golfarelli, Stefano Rizzi, and Andrea Proli. Designing What-if Analysis: Towards a Methodology. In *DOLAP*, pages 51–58, 2006.

[MP13] Stephan Müller and Hasso Plattner. Aggregates Caching in Columnar In-Memory Databases. In *IMDM @ VLDB*, pages 62–73, 2013.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *ACM SIGMOD*, pages 1–2, 2009.

[Pla14] Hasso Plattner. The Impact of Columnar In-Memory Databases on Enterprise Systems. *PVLDB*, 7(13):1722–1729, 2014.

[WBB+03] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, and Sankar Subramanian. Spreadsheets in RDBMS for OLAP. In *ACM SIGMOD*, pages 52–63, 2003.

[WBB+05] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Aman Naimat, Lei Sheng, Sankar Subramanian, and Allison Waingold. Query by Excel. In *VLDB*, pages 1204–1215, 2005.

[Zwi03] E. Zwicker. *Prozeßkostenrechnung und ihr Einsatz im System der integrierten Zielverpflichtungsplanung*. Techn. Univ. Berlin, 2003.

Panel: „Big Data – Evolution oder Revolution in der Datenverarbeitung?“

Panel: „Big Data – Evolution oder Revolution in der Datenverarbeitung?“

Alexander Böhm, Mathias Golombek, Christoph Heinz, Henrik Löser, Alfred Schlaucher

Moderation: Thomas Ruf

1 Abstract

Das Schlagwort Big Data hat in den letzten Jahren große Popularität gewonnen. Andererseits befassen sich Datenbanksysteme seit ihrer Entstehung mit großen Datenmengen, d.h. mit der Verwaltung von Datenmengen, die hart an der Grenze des technisch Beherrschbaren liegen. Mit der Verschiebung dieser Grenzen nach oben und der Steigerung der Anforderungen an Verarbeitungskomplexität und geschwindigkeit entstanden in der Datenbankwelt immer neue Lösungsansätze und Konzepte, wie z.B. verteilte und föderierte Datenbanksysteme, NOSQL-Systeme, Hauptspeicherdatenbanksysteme, aktive Datenbanksysteme und Datenstromanalysesysteme usw. Ist die neue Aufmerksamkeit für das Thema Big Data also tatsächlich durch neue Aspekte gerechtfertigt, oder bekommt ein lange schon adressiertes Thema endlich die notwendige Aufmerksamkeit? Sind die vielbeschworenen 3-.5 „V“ (volume, variety, velocity, ggf. auch veracity und value) tatsächlich revolutionär neue Gesichtspunkte? Muss man sich je nach Art der Daten (strukturiert, semistrukturiert, unstrukturiert) für andere Systeme entscheiden, oder gibt es für Big Data eine Gesamtlösung statt eines Zwangs zum „entweder – oder“ bei der Wahl zwischen NOSQL, relationalem Datenbanksystem, Event Processing, usw.? Diese Fragen werden im Panel adressiert werden; Vertreter verschiedener kommerzieller Anbieter stellen sich der Diskussion.

2 Biografien

Alexander Böhm ist Datenbankarchitekt bei SAP. Er arbeitet dort am Hauptspeicherdatenbanksystem HANA mit Fokus auf Performance-Optimierung. Zuvor promovierte er an der Universität Mannheim, wo er an der Entwicklung effizienter und skalierbarer Applikationen arbeitete.

Mathias Golombek ist seit November 2013 Teil des Vorstands der EXASOL AG und verantwortet den Bereich Technologie des Unternehmens. Zuvor war der studierte Informatiker als Head of Research & Development des Anbieters analytischer Plattformen tätig. Golombek begann seine Karriere 2004 als Software-Entwickler für EXASOL. Bevor er 2006 die Verantwortung für den Bereich Forschung und Entwicklung übernahm, war er Leiter des Teams für Datenbankenoptimierung.

Christoph Heinz ist in der R&D der Software AG im Bereich Intelligent Business Operations tätig und hat unter anderem in den Gebieten Complex Event Processing und Visual Analytics gearbeitet. Zuvor war er bei RTM Realtime Monitoring GmbH für Marketing und Vertrieb verantwortlich. Im Rahmen seiner Forschungsaktivitäten an der Philipps-Universität Marburg hat sich Christoph Heinz mit der Anwendung statistischer Verfahren im Complex Event Processing-Umfeld auseinandergesetzt.

Henrik Loeser ist Enablement Specialist und Software Deployment Project Executive bei IBM Analytics im Böblinger IBM-Labor. Er hilft Kunden und Partnern beim Einsatz von DB2 und anderen Information Management- und Analytics-Produkten. Zuvor war er Software-Entwickler und Technischer Leiter in den USA und Deutschland. Studiert und promoviert hat er an der Universität Kaiserslautern. Henrik Loeser ist von der Open Group als Distinguished IT Specialist zertifiziert. In seiner Freizeit lehrt er Datenmanagement am Campus Friedrichshafen der Dualen Hochschule Baden-Württemberg Ravensburg und bloggt zu DB2- und anderen IT-Themen.

Thomas Ruf ist freier IT-Berater für die Themengebiete Big Data, Business Intelligence und Data Warehousing. Vor seiner selbständigen Tätigkeit war er lange Zeit bei der GfK Retail and Technology GmbH in Nürnberg beschäftigt und leitete dort als Geschäftsführer den Bereich Research & Development. Neben seinen beruflichen Aktivitäten war und ist Prof. Ruf an der Universität Erlangen-Nürnberg tätig, wo er seit 2006 eine außerplanmäßige Professur für Praktische Informatik innehat. Er ist Mitglied der Leitungsgruppe der GI-Fachgruppe Datenbanksysteme und Mitherausgeber der Zeitschrift „Datenbank Spektrum“. Thomas Ruf ist Autor zahlreicher wissenschaftlicher Publikationen und zweier Bücher des Fachgebiets Angewandte Datenbank-Technologie.

Alfred Schlaucher startete Mitte der 1980er Jahre mit Datenbankprogrammierung auf Großrechnern von Siemens und IBM. In den 1990er Jahren beschäftigte er sich mehrere Jahre mit Datenmodellierung und Metadatenmanagement und seit Mitte der 1990er mit Data Warehouse. Seit Ende der 1990er ist er für Oracle im Bereich Data Warehouse aktiv.

Demoprogramm

AutoShard – A Java Object Mapper (not only) for Hot Spot Data Objects in NoSQL Data Stores

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Andreas Thor
Deutsche Telekom, University of Applied Sciences for Telecommunications
thor@hft-leipzig.de

Abstract: We demonstrate AutoShard, a ready-to-use object mapper for Java applications running against NoSQL data stores. AutoShard's unique feature is its capability to gracefully shard hot spot data objects that are suffering under concurrent writes. By sharding data on the level of the logical schema, scalability bottlenecks due to write contention can be effectively avoided. Using AutoShard, developers can easily employ sharding in their web application by adding minimally intrusive annotations to their code. Our live experiments show the significant impact of sharding on both the write throughput and the execution time.

1 Introduction

NoSQL data stores are highly appealing in web development, where applications require high scalability and high availability. Their flexible data model suits an agile software development style since the database schema does not have to be designed up front. Moreover, the sheer scalability of these systems is impressive: Due to a highly distributed architecture, they gracefully handle large amounts of users and data. To optimize user response time, several NoSQL data stores implement optimistic concurrency control [ADE12]. Transactions execute immediately, and then check at commit time if some other transaction has already committed a conflicting update. If so, the transaction is aborted and it is up to the application to handle the failure. This approach is appropriate for web applications where users spend most of their time browsing data, rather than generating content.

Since web applications are commonly read-intensive, users are less likely to concurrently update the same data. However, there are certain features in web applications that are inherently prone to write contention: This could be something as simple as a counter registering the number of visitors to a website, or the number of votes (or "likes") cast interactively against a blogpost. Write contention on *hot spot data objects* has a long legacy in database research, and at the same time is a timeless topic, e.g., recently motivating sophisticated concurrency control protocols that avoid shared-memory writes [TZK+13].

The established approach in the developer community for managing hot spot data objects in NoSQL data stores is *property* or *counter sharding* [ST14, San12] on the level of the

database schema. Sharding physically balances write requests by dividing logical units of data into multiple shards. For example, instead of storing a *single* counter we maintain *multiple* shard counters. Updates are performed on one shard (chosen at random) and the sum over all shards is the overall counter value. While intuitive to grasp, a robust implementation of sharding is nevertheless a nontrivial task: On top of aggregated reads and distributed writes, sharding has to be safe under transactions. Custom-coding sharding requires a deep understanding of the underlying concurrency control mechanisms, is system-dependent, and introduces additional complexity to the application code.

In this demonstration, we present *AutoShard*, a Java object mapper designed for NoSQL document stores of the Google Cloud Datastore [San12] family. AutoShard relieves the application developer from handling low-level sharding, and thus restores a clearer separation between logical and physical database design. We have shown in [ST14] that AutoShard significantly improves throughput and execution times of writes against hot spot data objects. AutoShard is easy-to-use, since it merely requires declarative annotations in Java classes and relies on self-modifying code to handle sharding transparently.

2 The AutoShard Object Mapper

The AutoShard object mapper is, to our knowledge, the first Java object mapper capable of sharding data automatically, based on simple annotations. Like other NoSQL object mappers, AutoShard takes care of the mundane marshalling of persisted entities into Java objects and back, thus greatly simplifying application development. In the following, we show how AutoShard addresses write contention. The Java class from Figure 1 (left) represents a blog post that can be voted up. As customary with object mappers, annotation `@Entity` specifies that an instance of class `BlogPost` is persisted as an entity. Annotation `@Id` marks the unique key.

The `votes` counter is a performance bottleneck when several users concurrently vote on the same blogpost. To solve this problem, we merely add the annotation `@Shardable` to declare that counter `votes` is to be sharded. When processing shards, the method annotated with `@ShardMethod` will be applied to a single shard, rather than the global value of the votes counter. We could even declare several sharding functions (e.g., to increment and to decrement votes). Since in this example the shard method is incrementation, we specify zero as the neutral element (see `neutral=0`). This is exploited in initializing new shards. With annotation `@ShardFold`, we declare the static function `sum` as the folding function. This function is called for aggregating over all shards. We may specify even more complex folding operations, as long as they are commutative and associative. It is the responsibility of the developers to correctly annotate their Java classes.

The AutoShard approach relies on self-modifying code, by blending Java code with Groovy technology (*www.groovy.codehaus.org*). Groovy is a dynamic language that runs in the JVM and smoothly inter-operates with Java code. Further, Groovy allows us to annotate code structures for transformations in the abstract syntax tree (AST) during compilation. Figure 1 (middle) shows the architecture of the AutoShard framework. A Java

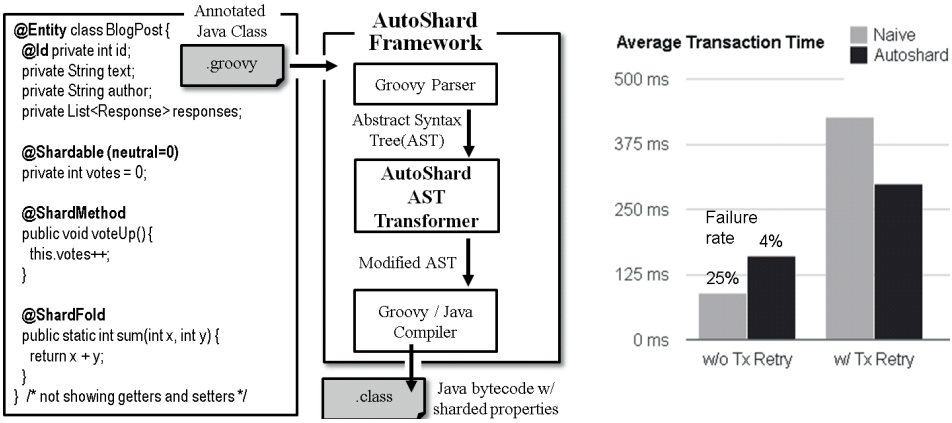


Figure 1: Compilation of an annotated Java class (left) with the AutoShard framework (middle). Evaluation of property sharding with 2,0000 users incrementing 16 counters (right).

class with AutoShard annotations serves as input. The Groovy parser produces an AST and our AutoShard AST transformer restructures this tree. Class members annotated as `@Shardable`, as well as the sharding and the folding method, are now transformed. We refer to [ST14] for the details of this compilation, and merely present the basic idea here.

For the sharded property `votes`, AutoShard's compilation introduces a new (private) attribute `shard_votes` that stores a single shard value. The body of user-defined method `voteUp` is transferred to a private method, and the original method is replaced. This new implementation calls the original function both for the shard value (`shard_votes`) and for the actual value (`votes`). Since the signatures of the class methods do not change, the remaining application code need not be adapted.

When a persisted `BlogPost` is loaded, AutoShard retrieves the main entity to map all unsharded class members. For the sharded class member `votes` it reads all shards and generates *two* data members. First, the (regular) data member `votes` is initialized to the aggregated shard value. AutoShard uses the `@ShardFold` method (`sum` for class `BlogPost`) to aggregate over all shards. Second, the (internal) member `shard_votes` is initialized to the neutral element zero. When shard method `voteUp` is invoked for updating the counter value, the update is executed on both the (regular) data member `votes` as well as the (internal) data member `shard_votes`. This ensures that whenever the application code accesses `votes`, it sees the expected value.

When entity `BlogPost` is persisted after changes have been made, AutoShard first updates the main entity (if necessary). For the sharded class member, a random shard is loaded from storage. Its value is updated by invoking the `@ShardFold` method (`sum` for class `BlogPost`) on the loaded shard value and on `shard_votes`. The shard is persisted within a nested transaction, so we do not interfere with transactions that may be running in the remaining code. Since the sharded value is re-set to the neutral element, it will capture future updates. The regular property `votes` still holds the current value.

3 Demonstration Description

In our demonstration we illustrate how AutoShard can effectively be employed in building scalable web applications that persist data in NoSQL data stores. Our scenario deals with a Java implementation of a voting tool. The application runs on Google App Engine and is backed by Google Cloud Datastore. We start with a naive implementation that does not take any precautions w.r.t. massively concurrent writes. We simulate an increasing number of users voting on blogposts. This causes write contention, and ultimately, failed web requests. We dynamically visualize the success rate as well as the average transaction time using Google Charts. For instance, Figure 1 (right) shows the effects of sharding counters in a voting app where 2,000 users concurrently update 16 counters.

We then add a few lines of annotations to the Java class declarations within Eclipse. Upon the push of a button, the Java code is re-compiled and deployed to Google App Engine. To the interested audience we can also go into the implementation details of the AutoShard AST Transformer. We repeat the experiment with the AutoShard-powered application. Now, the application handles web requests much more successfully: In the example from Figure 1, with the naive implementation, 25% of all transactions fail. Generating 16 shards with AutoShard, the failure rate can be reduced down to 4%.

In a second scenario, we add a simple retry mechanism and repeat failing transactions until they succeed. This simple approach obviously increases the chances of requests succeeding at the expense of higher execution times. We re-run our experiment with the naive and the sharded approach. Now, all requests succeed eventually, yet the sharded version shows superior transaction times (see Figure 1 (right)).

The audience may further investigate the details and differences between the naive and the sharded version by manipulating certain parameters such as the number of shards, or by employing alternative sharding strategies [ST14]. Again, AutoShard's easy-to-use annotation approach makes it convenient to modify the Java source code that is then compiled and deployed to Google App Engine. The audience will be able to assess the trade-offs between sharding strategy, success rate, and execution time under different workloads.

Acknowledgement: The authors would like to thank Werner Schier for contributing to the AutoShard code base as part of his Master thesis project at OTH Regensburg.

References

[ADE12] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. *Data Management in the Cloud – Challenges an Opportunities*. Synthesis Lectures on Data Management. 2012.

[San12] Dan Sanderson. *Programming Google App Engine*. Google Press, 2012.

[ST14] Stefanie Scherzinger and Andreas Thor. AutoShard - Declaratively Managing Hot Spot Data Objects in NoSQL Data Stores. In *Proc. WebDB'14*, 2014.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, et al. Speedy Transactions in Multicore In-memory Databases. In *Proc. SOSP'13*, 2013.

AutoStudio: A Generic Web Application for Transforming Dataflow Programs into Action

Nikhil-Kishor Rane, Omran Saleh

Technische Universität Ilmenau, Ilmenau, Germany
{first.last}@tu-ilmenau.de

Abstract:
In this paper, a user-friendly, interactive, and easy-to-use web-based application, is introduced. "AutoStudio" is a generic application where the users can generate dataflow programs for different dataflow languages, such as PipeFlow and Pig, using a drag and drop functionality. Using this application, the users can assemble the operators that comprise a particular program without being an expert in the target language and without his/her awareness of its operators and constructs. Therefore, the rapid development can be achieved. Our application also provides additional functionalities which makes it a "One-Stop-App" such as compiling, saving, executing, and checking the status of the generated dataflow programs.

1 Introduction

In today's world, an increasing amount of data is becoming obtainable over the internet. With the growth of systems focusing on the data processing, there is always a requirement of devising innovative ways of processing more data in less time, i.e., the big data challenge. Recently, several platforms have been developed to address this challenge. Typically, some of these platforms, e.g., PipeFlow [SBS15] and Pig[1], provide a declarative interface in the form of dataflow specifications where the processing steps are described by programs or scripts formulated as a sequence of statements. Each statement declares an operator which defines a specific task for manipulating the data, i.e., tuples. The flow of data is illustrated as lines or pipes between the operators. By using the output pipe of one operator as input pipe of another operator, a dataflow graph is formed.
However, the user currently has to manually perform all tasks of creating the dataflow script, editing, and compiling. Though, several text-based plug-in editors have been developed, e.g., pig-eclipse[2] and pig-mode[3] which are dedicated only for Pig[4] language, these editors focus mainly on syntax highlighting, code completion, and indentation. Namely, if the user needs to create and edit a particular dataflow script then he/she should be well versed with the syntax and programming constructs of the language. Therefore, no rapid development is possible as user needs to proceed by writing each programming statements

[1] http://pig.apache.org
[2] https://code.google.com/p/pig-eclipse/
[3] https://github.com/motus/pig-mode
[4] PipeFlow did not have any editor

correctly. Additionally, in these conventional approaches, concurrency is not supported and many users will not be able to simultaneously use the application. Moreover, the user also needs to manually check the status of programs in execution, which becomes cumbersome as the number of programs increases. This manual approach could lead to time and effort wastage if any errors or misconfigurations occur. These limitations motivated us to design a generic application to automate the whole process of generating and executing different dataflow languages; and as a result, **AutoStudio** was implemented.

2 AutoStudio System

The platform that we envision should be used by different people and organizations. It needs to be deployed only once and each user can access the application, create, and execute different dataflow scripts, such as PipeFlow and Pig, remotely from any location. With the advent of Internet, web application development has gained popularity. Web applications, presently, started substituting desktop applications for reasons of portability and usability. Thus, we have developed the **AutoStudio**. It is a very user friendly, interactive, and easy to use web application to run cross-platform using HTML5, Draw2D touch[5], and node.js[6]. It also relies heavily on JavaScript Object Notation (JSON) for data exchange and pre-compiled templates "hogan.js"[7]. AutoStudio's services and components are mentioned in Sect. 3 and Sect. 4, respectively. To perform any task, a user has to login to AutoStudio with a user ID and password. Once the user is successfully logged in, the user's home page is displayed with a list of **apps** and **flow-designs** sorted in the order they were last accessed as shown in Figure 1. This is the point where the **workflow** begins. Below is a brief description of the terms which make up AutoStudio.

Applications or "Apps": App is AutoStudio's terminology of referring to the design part of a workflow. An app defines a set of operators, connection types, properties, parameters, etc., which have pre-defined semantics. The semantics are later applied to a flow-design to transform it into a target language script. For instance, the apps which focus on PipeFlow and Pig scripts are PipeStudio and PigStudio, respectively.

Flow-design: The diagram a user creates in AutoStudio which is later transformed into a script is known as a flow-design. A flow-design corresponds to a specific app having pre-defined semantics. AutoStudio allows CRUD[8] operations on a flow-design and stores it primarily as JSON on the server. However, it can also be exported in JSON, SVG or PNG formats from AutoStudio at the click of a button.

Workflow: The point at which a user launches an app and starts creating a flow-design to the point when it actually executes on the server is called as workflow.

[5] `http://www.draw2d.org/draw2d/index.html`
[6] `http://www.nodejs.org`
[7] `http://twitter.github.io/hogan.js/`
[8] Create, read, update, and delete

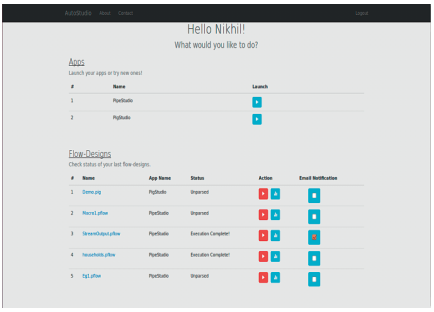


Figure 1: AutoStudio home page

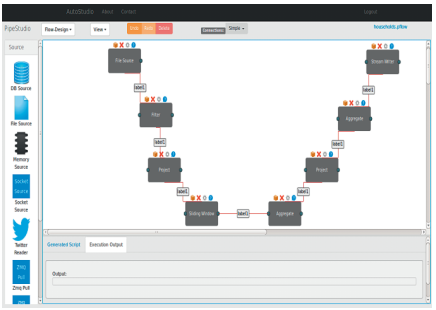


Figure 2: AutoStudio dashboard

3 AutoStudio Services

AutoStudio application provides several functionalities which makes it a "One-Stop-App" to handle the complete workflow. It enables users to leverage the emerging dataflow languages graphically via a collection of operators which could be "dragged and dropped" onto a drawing canvas (as in Figure 2). These operators are represented by icons. The user can assemble the operators in order to create a dataflow graph for a particular language in a logical way and visually show how they are related, and from this graph, equivalent script can be generated. This makes the user to be not aware of the language syntax and its constructs. Based upon the user's selection of apps from the home page, the right operators will be shown. By clicking on the operator icon, a pop-up window appears to let the user specify the parameters of operators, which are required. Moreover, the user can display the help contents for each operator. Besides these functionalities, the application also has a feature of script execution via calling the respective engine and displaying script execution result instantly and in real-time as well as emailing the user when the execution is complete. The application provides the options of saving the generated scripts or flow-designs for future reference, loading the saved script, and executing it whenever required.
Adding new operators: AutoStudio relies on JSON data for construction of its apps. Each app in our application is modeled in a single JSON file. Therefore, adding an operator, connection type or a container involves only adding the required {key:value} pairs in the app's JSON file. In our example of PipeStudio, this file is called *pipestudio.json*.
Adding new apps: Since our application is designed to be generic, a few components are needed in order to add a new app (currently, these components can be added manually, later, an interface can be used to configure the new app). These components are as follows:
1. *JSON file*: contains all configurations of operators, connection types, and containers.
2. *Resources*: all the required resources for the app like images, properties dialogs, etc.
3. *Database*: a database entry is required to make AutoStudio aware of this new app.
The flow-designs created using AutoStudio are stored in separate database collections. Hence, each app has its own collection which provides flexibility to apps to handle their flow-designs as per requirement. 4. *Parser*: to parse Draw2D touch output into a target language. For instance, in PipeStudio, the parser parses the output into PipeFlow language.
5. *Executor*: a shell script or command to run the parsed dataflow script.

4 AutoStudio Components

AutoStudio prominently uses open source softwares and frameworks. It should work out of the box in any modern browser on any platform supporting HTML5. The components are divided into two parts. The first part is the client side which uses HTML5, JavaScript, Cascading Style Sheets (CSS), jQuery (and helping libraries), twitter bootstrap, and hogan.js. Mostly, they are used for building the graphical user interface, performing Ajax requests, file uploading and downloading, etc. AutoStudio extensively uses pre-compiled hogan templates where the data returned from the server is simply passed to these templates for quick rendering. In addition, Draw2D touch is used to enable creation of diagram applications in a browser by creating and manipulating operators and connections.
The second part is the server side which consists of the node.js web server and a database. The logic on the server side is completely implemented using node.js and its supporting modules. The event-driven, non-blocking I/O model of node.js makes it fast, light-weight, and efficient. It is greatly suitable for data-intensive real-time applications. Several node.js modules are used. *Nconf* is used as an object-store for easy storage and retrieval of configuration properties for AutoStudio, its apps, parsers, etc. *Nodemailer* is used to send emails from node.js. Our application uses this module to send notifications to registered email IDs when the execution of a flow-design is complete. *Socket.io* is a real-time framework server for node.js and used extensively to send real-time stats to the client when a flow-design is in execution. On the database side, the most popular No-SQL database *MongoDB* is used. It is an open source, JSON-style document oriented, and supports conventional as well as additional properties for data-intensive applications. It is used to store and retrieve the users and the saved flow-designs information.

5 Demonstration

During the demo session we will bring a computer running our application and demonstrate its capabilities, ease-of-use, generics, and supported-services as above-mentioned by employing real examples. The presenter will show how the users can navigate through different apps in AutoStudio. He will create different dataflow programs for different languages by dragging and dropping the operators. Furthermore, he will compile and execute these scripts by using our application. Thus, the audience can see real-time results and check the status of the running programs. Additional functionalities also will be shown such as saving and loading the scripts as well as emailing the user when the execution is complete. Moreover, the presenter will demonstrate how to extend any of the available apps, e.g., using PigStudio or PipeStudio as backend, and add new operators on-the-fly.

References

[SBS15] Omran Saleh, Heiko Betz, and Kai-Uwe Sattler. Partitioning for Scalable Complex Event Processing on Data Streams. In *New Trends in Database and Information Systems II*, pages 185–197. 2015.

Cleager: Eager Schema Evolution in NoSQL Document Stores

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Abstract: Schema-less NoSQL data stores offer great flexibility in application development, particularly in the early stages of software design. Yet over time, software engineers struggle with the heavy burden of dealing with increasingly heterogeneous data. In this demo we present Cleager, a framework for eagerly managing schema evolution in schema-less NoSQL document stores. Cleager executes declarative schema modification operations as MapReduce jobs on the Google Cloud Platform. We present different scenarios that require data migration, such as adding, removing, or renaming properties of persisted objects, as well as copying and moving them between objects. Our audience can declare the required schema migration operations in the Cleager console, and then verify the results in real time.

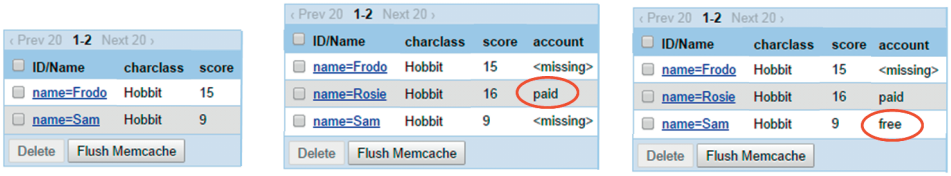
1 Schema Evolution in NoSQL Document Stores

This work concerns the maintainability of feature-rich, interactive web applications from the view-point of schema evolution. In particular, we target web applications backed by schema-less NoSQL document stores such as MongoDB [Mon14] or Google Cloud Datastore [San12]. This is an increasingly popular software stack: The programming APIs are easy to use, there is little setup time required with *database-as-a-service* offerings, and pricing is reasonable. Another sweet spot is that the schema does not have to be specified in advance, so developers may freely adapt the data structures as the application evolves.

Yet over time, dealing with an increasingly heterogeneous collection of persisted objects becomes a nuisance. Since today's schema-less NoSQL data stores still lack the proper tools for managing schema evolution, even the most basic *schema modification operations* (by the terminology of [CMDZ13]), such as adding or removing a field from persisted objects in batch, require programmatic solutions and therefore custom coding.

Example. We consider an online role playing game hosted on Google App Engine and backed by Google Cloud Datastore [San12]. Figure 1a shows the persisted player objects for Frodo and Sam. Each player has a unique *name*, further a *character class* and a *score*. The next release introduces paid accounts. Figure 2a shows the new object mapper class declaration with an *account* property. The annotation `@Id` declares the identifying property and `@Index` declares that a property can be queried (using Objectify [Obj14] syntax).

Figure 1b then shows the data instance after second-generation player Rosie has been



(a) First-generation players. (b) Adding a second-generation player. (c) After lazy migration of player Sam.

Figure 1: Screenshots from the Datastore Viewer illustrating the evolution of persisted objects (termed "entities") in Google Cloud Datastore over time.

```
@Entity
public class Player {
  @Id private String name;
  private String charclass;
  private int score;
  @Index private String account;

  /* Lazy migration after loading. */
  @OnLoad void onLoad() {
    if (account == null)
      account = "free";
  }
  /* Omitting getters and setters. */
}
```

(a) Java class declaration with Objectify annotations.



(b) Query evaluation on the players from Figure 1b.

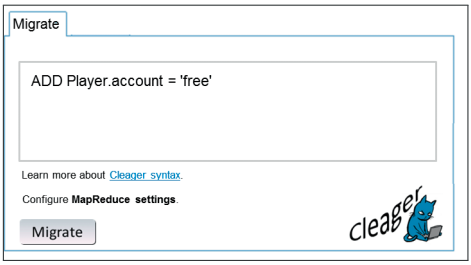
Figure 2: (a) Lazily introducing the new property *account* and (b) querying for free accounts.

persisted. As a schema-less NoSQL data store, Datastore allows for both generations of players to co-exist. Thus, the new code can be released without having to immediately migrate the legacy players Frodo and Sam. While this allows for rapid releases, it adds complexity to the application code. For instance, to search for all players with a free account, the GQL query (Google Query Language) in Figure 2b will return no results on the data instance from Figure 1b, since no persisted object satisfies the query predicate.[1]

One (partial) solution to this problem is *lazy* schema evolution through life-cycle annotations in object mapper class declarations, an increasingly popular approach [SHKS15]. When legacy player Sam is loaded into the application space with the class declaration from Figure 2a, the `@OnLoad`-method is executed. This sets a default value for the new attribute *account*. After the changes have been persisted, Sam's player object has been successfully migrated (c.f. Figure 1c). Yet unless Frodo's object is loaded into the application space, the missing property will not be added.

To reliably evaluate this query, *all* legacy objects need to be migrated. Today, developers actually write custom code to *eagerly* migrate legacy objects. Writing custom code is time consuming and requires careful testing, since the code is executed directly on production data. The Cleager framework offers a new and declarative alternative for eager migration.

[1] In GQL, it is actually *not possible* to formulate a query that returns players Frodo and Sam by a SQL-style query predicate such as "`where account is not null`". GQL queries are evaluated over indexes, and missing values are not indexed. We refer to [SKS13] for a brief overview over *null* values in NoSQL data stores.



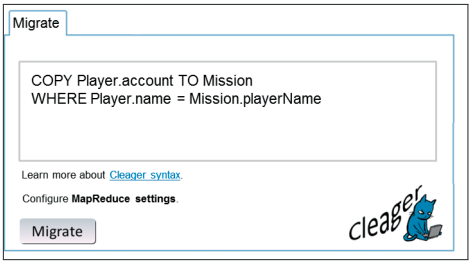
(a) Adding a property in the Cleager console.

```
def map(String key, JSON value) {
 if value is a Player object
 then value.addProperty("account", "free");

 put(key, value); // Persisting the change.
}

/* A map job is sufficient. */
```

(b) The MapReduce job for adding a property.



(c) Copying a property in the Cleager console.

```
def map(String key, JSON value) {
 if value is a Player object
 then emit(value.name, value);

 if value is a Mission object
 then emit(value.playerName, value);
}

def reduce(String key, List<JSON> valueList) {
 let p be the Player object in valueList;
 for each Mission m in valueList do
  m.setProperty("account", p.account);
  put(m.key, m); // Persisting the change.
 od
}
```

(d) The MapReduce job for copying a property.

Figure 3: The Cleager console with schema modification operations ((a) and (c)) for migrating first-generation players. The declarative operations are then executed as MapReduce jobs ((b) and (d)).

2 MapReduce-powered Eager Data Migration with Cleager

In [SKS13] we have proposed a declarative *NoSQL schema evolution language* by which developers can specify common schema modification operations, such as adding, deleting, or renaming properties in batch. This covers the bulk of operations frequently required in web development, according to a study in [CMTZ08]. In addition, our language allows that properties are moved or copied between objects, since data duplication and denormalization are fundamental principles in NoSQL data stores. This goes beyond the schema modification operations commonly supported by relational databases. For more complex and infrequent modifications (e.g., splitting or merging property values), developers can always resort to custom MapReduce jobs. We refer to [SKS13] for the full specification of the Cleager language.

Figure 3a shows the Cleager console with a declarative statement that adds an *account* property to all players. Cleager executes this migration as a MapReduce job. For adding, removing, and deleting properties from persisted objects, a simple Map jobs suffices. Figure 3b shows the MapReduce job for adding an *account*-property in pseudo-code. Here, the reducer implements the identity function. This eager migration is to be executed on the legacy player objects prior to the new release, which introduces paid accounts.

Figure 3c shows the declaration for copying property *account* from the player object to all

missions belonging to this player. Since Google Cloud Datastore, like most NoSQL data stores, does not support joins in its query language, data denormalization is a common practice to avoid having to implement joins at runtime within the application code. In Figure 3d, we show the pseudo-code of the corresponding MapReduce job.

Cleager thus translates declarative schema migration operations into MapReduce jobs, making use of massive parallelization on the Google Cloud Platform infrastructure [Goo14].

3 Demo Outline

1. We present a small gaming application running as software-as-a-service on Google AppEngine backed by Google Cloud Datastore. Our audience can inspect the player data in the Datastore Viewer. We then show how changes to the application code cause heterogeneous player objects to co-exist in the data store, as in Figure 1.
2. We show that lazy schema evolution via object mapper libraries only affects those objects that are loaded into the application space at application runtime.
3. We show how queries over heterogeneous data can lead to unexpected results, since legacy objects may not be considered in query evaluation, as depicted in Figure 2b.
4. Our audience may write eager schema migration operations using our language from [SKS13], within the web-based Cleager console (e.g., as seen in Figure 3a).
5. We monitor the execution of the schema migration operations as MapReduce jobs in the Google Cloud Platform on large data sets. Together, we then inspect the results.

Acknowledgements: We thank Kok Wing Tang from OTH Regensburg for implementing a first prototype of Cleager as part of his Bachelor thesis.

References

[CMDZ13] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. *The VLDB Journal*, 22(1):73–98, 2013.

[CMTZ08] Carlo Curino, Hyun Jin Moon, Letizia Tanca, and Carlo Zaniolo. Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In *Proc. ICEIS'08*, 2008.

[Goo14] Google Inc. Google Cloud Platform, November 2014. `https://cloud.google.com/`.

[Mon14] MongoDB, November 2014. `http://www.mongodb.org/`.

[Obj14] Objectify, November 2014. `https://code.google.com/p/objectify-appengine/`.

[San12] Dan Sanderson. *Programming Google App Engine*. O'Reilly Media, Inc., 2012.

[SHKS15] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. Schemaless NoSQL Data Stores - Object-NoSQL Mappers the Rescue? In *Proc. BTW'15*, 2015.

[SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. DBPL'13*, 2013.

Datenanalyse in den Digital Humanities – Eine Annäherung an Kostümmuster mittels OLAP Cubes

Michael Falkenthal[1], Johanna Barzen[1], Simon Dörner[2], Vadym Elkind[2], Jan Fauser[2], Frank Leymann[1], Tino Strehl[2]

Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Nachname@iaas.uni-stuttgart.de[1]

Hochschule Reutlingen
Herrmann Hollerith Zentrum
Alteburgstr. 150
72762 Reutlingen
Vorname.Nachname@student.reutlingen-university.de[2]

Abstract: Im Film ist das Kostüm eines der prominentesten Gestaltungselemente, um Aussagen über eine Rolle, deren Charakter, Stimmung und Transformation, wie auch über Ort- und Zeitgegebenheiten zu kommunizieren. Durch Kostümmuster sollen Kostümbildner befähigt werden, effizient auf bewährte Designlösungen zurückgreifen zu können. Diese Demo zeigt, wie generelle Designprinzipien aus einer großen Anzahl an Kostümen aus Filmen für die Entwicklung dieser Kostümmuster mittels OLAP Cubes abstrahiert werden können. Um generelle Designprinzipien feststellen zu können, werden Kostüme über kategoriale Merkmalstaxonomien beschrieben und in verschiedenen Aggregationsstufen ausgewertet. Die Abstraktion von generellen Lösungen für Kostümmuster wird durch Drill-Down und Roll-Up Mechanismen unterstützt.

1 Einleitung und Motivation

Muster und Mustersprachen sind in vielen Domänen ein weit verbreitetes Mittel, um erprobtes Lösungswissen zu wiederkehrenden Problemen zu erfassen und dieses Wissen effizient für andere nutzbar zu machen. Aus der Architektur kommend [AIS77], hat sich das Konzept des Musters in der Informatik weit verbreitet. Dies lässt sich durch die Vielzahl an Publikationen aus den verschiedensten Bereichen der Informatik belegen, wie z.B. Cloud-Applikationsarchitektur, Unternehmens-IT-Architektur, Green-IT, Applikationssicherheit oder Benutzerschnittstellendesign, um nur einige zu nennen.

Aber auch in anderen Domänen, wie der Pädagogik oder der Produktentwicklung, hat das Konzept des Musters erfolgreiche Anwendung gefunden. Auch in der Domäne von Kostümen in Filmen erscheint der Einsatz von Mustern vielversprechend, indem adäquate textile Ausdrücke für Charaktere, die einfach vom Rezipienten verstanden werden (wie kommuniziert man beispielsweise einen guten Sheriff gegenüber einem bösen Sheriff mittels eines Kostüms?) als Lösungen wiederkehrender Designprobleme eines Kostümbildners in Mustern abgefasst werden [SBL12]. Als ein Projekt der Digital Humanities, bei dem sich geisteswissenschaftlichen Fragestellungen mittels neuer Methoden aus der Informatik genähert werden soll, wird durch interdisziplinäre

Forschung von Informatikern und Geisteswissenschaftlern eine Mustersprache für Kostüme im Film entwickelt [SBL12]. Basierend auf einem repräsentativen Fundus an Filmen (60 Filme mit jeweils ca. 200 Kostümen) wird eine Datenbasis – auch Lösungsrepository genannt [FBF14] – angelegt, in der Kostüme durch relevante Parameter, organisiert in Taxonomien, beschrieben werden [Ba13]. Jedes Kostüm in dieser Datenbasis repräsentiert eine konkrete Lösung für ein Designproblem. Im Gegensatz zu konkreten Lösungen im Lösungsrepository beinhalten Muster abstrahierte, generelle Lösungsprinzipien [FBB14]. Für die Formulierung von Mustern muss daher die Quintessenz aus einer Menge von Kostümen mit gleichem Effekt bezüglich eines Designproblems extrahiert werden [BL14].

In der Demo wird gezeigt, wie die Datenbasis mit Hilfe von Online Analytical Processing (OLAP) Cubes ausgewertet werden kann, sodass Muster aus der Menge an konkreten Lösungen abstrahiert werden können. Dazu wird eine Werkzeugumgebung vorgestellt, die eine automatisierte Aufbereitung und Übertragung der Daten aus der Datenbasis in ein Data Warehouse (DWH) ermöglicht. Weiterhin wird gezeigt, wie das Datenbankschema des DWH aufgebaut sein muss, um performante OLAP Cubes für die Auswertung des Datenbestandes zu ermöglichen und wie die Analyse der Daten aus filmwissenschaftlichen Gesichtspunkten erfolgt.

Nach der einleitenden Motivation in diesem Abschnitt, wird das grundlegende methodische Konzept der Auswertung in Zusammenhang mit dem Datenmodell der auszuwertenden Kostüme in Abschnitt 2 beschrieben. In Abschnitt 3 wird die Architektur des zu demonstrierenden Systems erläutert, während diese Arbeit mit einer Beschreibung des Ablaufs der Demo in Abschnitt 4 schließt.

2 Datenmodell und methodisches Konzept

Dem vorgestellten Konzept liegt eine Datenbasis mit konkreten Designlösungen (Kostümen) zugrunde. Ein *Kostüm* besteht aus mindestens einem *Basiselement*, wobei ein Basiselement einem Kleidungsstück wie z.B. einem Hemd entspricht. Ein Basiselement setzt sich aus mehreren *Teilelementen* wie Hemdkragen oder Knopfleiste zusammen. Basis- und Teilelemente werden durch Attribute wie Farbe, Form, Material oder Zustand beschrieben, während zur Beschreibung eines Kostüms Attribute wie Charaktereigenschaft, Funktion, Zustand oder Körpermodifikation herangezogen werden. Damit Kostüme im Kontext von rollen- bzw. filmspezifischen Attributen ausgewertet werden können, werden sie mit genau einer *Rolle* verknüpft, welche wiederum genau einem *Film* zugeordnet ist. Eine Rolle wird durch Attribute wie Beruf, Alter, Familienstand, Geschlecht oder Stereotyp spezifiziert. Attribute wie Erscheinungsjahr, Genre, Produktionsort oder Stil werden zur Beschreibung eines Films verwendet. Eine komplette Auflistung der Attribute aller Entitäten und deren Semantik ist in [Ba13] zu finden.

Um wiederkehrende Kostümmuster abstrahieren zu können, müssen Ausprägungen von Attributen und Attributskombinationen der Kostüme in der Datenbasis einfach vergleichbar gemacht werden. Zu diesem Zweck werden Taxonomien vorgegeben, die

den Wertebereich der Attribute festlegen und eine einheitliche Begrifflichkeit bei der Beschreibung von Kostümen gewährleisten [Ba13]. Die Attribute einer Entität können somit als pivotierbare Dimensionen mit kategorialen Ausprägungen verwendet werden, um neue Sichten zur Analyse des Datenbestandes zu generieren. Zusätzlich definiert jede Taxonomie eine Hierarchie von Super- und Subtypen. Dadurch können Kostüme über den Vergleich von Attributskombinationen über unterschiedliche Abstraktionsstufen hinweg auf gemeinsame Ausprägungen hin untersucht werden.

Abbildung 1 zeigt exemplarisch ein Auswertungsszenario, in dem analysiert wird, wie oft Basiselemente des Typs Kleid eine spezifische Farbausprägung haben. Dafür werden Basiselementtypen vertikal und Farben horizontal angeordnet. Die Zahlen im rechten Teil der Abbildung entsprechen dem Auftreten der entsprechend vertikal bzw. horizontal zugehörigen Merkmalsausprägungen. So existieren im Beispiel insgesamt 4 Basiselemente des Typs Kleid mit der Farbe Rot sowie 3 gelbe und 2 blaue. Durch die hierarchische Struktur der Basiselementtypen kann die Analyse detailliert werden, indem durch einen Drill-Down angezeigt wird, wie viele Basiselemente der Subtypen Abendkleid und Sommerkleid entsprechend rot, gelb und blau sind. Damit wird ersichtlich, dass bei der Aggregation das Auftreten von Merkmalsausprägungen von Subtypen aufsummiert wird. Der höchste auswertbare Detaillevel ist mit den Blattelementen einer Taxonomie festgelegt, im Beispiel ist dies für die Basiselementtypen das schulterfreie Kleid und das One-Shoulder Kleid. Über dieses Prinzip der Aggregation von konkreten Typen hin zu abstrakteren Typen – Roll-Up genannt – können Zusammenhänge auf dem Level von Merkmalskategorien ausgewertet werden.

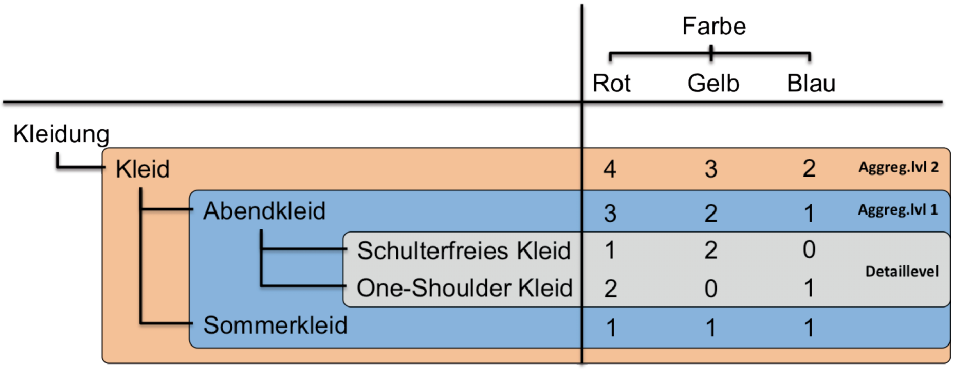


Abbildung 1: Aggregation über Kategorieebenen von Merkmalstaxonomien

3 Systemumgebung

Die in den ausgewählten Filmen vorkommenden Kostüme werden durch Domänenexperten betrachtet und über eine Webanwendung in eine MySQL Datenbank eingetragen. Mittels der Microsoft Integration Services werden die Daten aus der MySQL Datenbank in einem ETL-Prozess aufbereitet und in ein Microsoft SQL

Server 2012 DWH geladen. Das erstellte DWH verwendet das Galaxy-Schema. Somit können mehrere Faktentabellen erstellt werden, die gezielt auf bestimmte Analyseszenarien ausgelegt sind und nur die Szenario-relevanten Attribute enthalten. Über Microsoft Analysis Services werden OLAP Cubes bereitgestellt, die zur Auswertung des Datenbestands in Microsoft Excel eingebunden und visualisiert werden können.

4. Beschreibung der Demo-Session

In der Demo wird insbesondere darauf eingegangen, wie eine Werkzeugumgebung in einem Digital Humanities Projekt gestaltet werden kann, sodass neue informationstechnische Auswertungsmethoden für Geisteswissenschaftler bereitgestellt werden und gleichzeitig so viel informationstechnische Komplexität wie möglich für die Endanwender verborgen bleibt. Dafür umfasst die Demo alle Schritte vom Vorbereiten der Rohdaten bis hin zur Auswertung in Excel. Die wichtigsten Punkte des ETL-Prozesses werden erläutert, die zur Bereitstellung des DWH führen. Der Aufbau des DWH wird gezeigt. Anschließend wird die Einbindung und die Vorbereitung des DWH mittels Microsoft SQL Data Tools vorgestellt, in denen die Cube Einstellungen vorgenommen werden. Weiterhin zeigt die Demo wie der Datenbestand in Excel ausgewertet werden kann. Der Endanwender wählt die benötigten Attribute aus, um den Datenbestand mit bestimmten Fragestellungen zu untersuchen. Mittels Pivot-Tabellen können die Daten in jeder Dimension analysiert und als Pivot-Charts visualisiert werden. Durch das Auftreten von Spitzen in der Häufigkeitsverteilung der Pivot-Charts können erste Rückschlüsse auf mögliche Muster gezogen werden.

Literaturverzeichnis

[AIS77] Alexander, C.; Ishikawa, S.; Silverstein, M.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York, 1977.

[Ba13] Barzen, J.: Taxonomien kostümrelevanter Parameter: Annäherung an eine Ontologisierung der Domäne des Filmkostüms. Universität Stuttgart, Stuttgart, 2013.

[BL14] Barzen, J.; Leymann, F.: Costume Languages as Pattern Languages. In: Proceedings of Pursuit of Pattern Languages for Societal Change - Preparatory Workshop 2014, Krems, 2014 (akzeptiert).

[FBB14] Falkenthal, M.; Barzen, J.; Breitenbücher, B.; Fehling, C.; Leymann, F.: From Pattern Languages to Solution Implementations. In: Proceedings of the 6th International Conference on Pervasive Patterns and Applications, Venedig, 2014; S. 12–21.

[FBF14] Fehling, C.; Barzen, J.; Falkenthal, M.; Leymann, F.: PatternPedia – Collaborative Pattern Identification and Authoring. In: Proceedings of Pursuit of Pattern Languages for Societal Change - Preparatory Workshop 2014, Krems, 2014 (akzeptiert).

[SBL12] Schumm, D.; Barzen, J. ; Leymann, F.; Ellrich, L.: A Pattern Language for Costumes in Films. In: Proceedings of the 17th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, 2012. ACM Press, New York, 2012.

High Quality Information Delivery: Demonstrating the Web in Your Pocket for Cineast Tourists

Florian Stahl*, Adrian Godde†, Bastian Hagedorn†, Bastian Köpcke†,
Martin Rehberger†, Gottfried Vossen* † ‡
∗ WWU Münster, ERCIS, Florian.Stahl@ercis.de
† WWU Münster, CS Dept., {a.godde | b.hagedorn | basti.k | m.rehberger}@wwu.de
‡University of Waikato Management School, vossen@waikato.ac.nz

Abstract: The Web in Your Pocket is an information platform providing users with high quality content, sourced from the Web and curated by human experts. We describe the process as well as its implementation and elaborate on our demo case: tourists wishing to explore remote filming locations.

1 Introduction

Online information has exploded over the last decades. This implies that it becomes increasingly harder to find information – in the sheer mass of information available – that is really relevant to a particular situation: the proverbial needle in a haystack. This may be because of complex queries requiring data integration [Cer10] or because comprehensive semantic knowledge is necessary, yet not available [HSBW13]. Furthermore, keeping information available when being offline is currently not integrated in search processes. In some cases, this can enhance the perceived utility of the system tremendously.

To overcome this dilemma, we have developed a framework that does not answer queries solely based on a pre-assembled index, but based on a subject-specific database that is sourced from the Web, integrated and curated by domain experts, kept up to date automatically, and dynamically generated based on vast user input. Since our framework is supposed to make relevant information accessible on a mobile device when no Internet connection is available, we refer to it as Web in your Pocket (WiPo) [DSV12]. The contribution of this demo lies in presenting the implementation of a complex data gathering, processing, transforming, editing and curating process that enables users to find relevant information which can be persisted offline.

Imagine a tourist travelling New Zealand who wishes to visit locations where the Lord of the Rings films have been shot. Confessedly, Web sites such as *http://www.movie-locations.com/* provide information on filming locations, but they do so in an un-integrated way. Applying the generic WiPo framework to the situation of cineast tourists, we realised a platform on which information about filming locations is presented in a way that incorporates information about the film itself (e. g., sourced from Wikipedia), images, snippets, or film trailers, (e. g., sourced from Youtube), and general tourism information (e. g., sourced

from local tourism and accommodation Web sites). Furthermore, we provide a means to make this information available offline, so that tourists do have them at hand, when they are for example in a New Zealand region where no Internet connection is available. We mention that the system supports a number of business models. For the demo case a free-to-use model is most appropriate. The operating cost could either be covered in a Wikipedia-like fashion where people contribute time and money for the sake of the greater good. Alternatively, a third party with a severe interest in publishing the information, such as local tourism agencies, could step in as a sponsor.

To our knowledge, there are only a few approaches that combine curation and Web Information Retrieval. Sanderson et al. [SHL06] suggested to apply a procedure where predefined queries are sent to pre-registered services on a nightly basis. The retrieved information is audited by curators who decide upon its relevance. A similar harvest and curate approach was suggested by Lee et al. [LMH09] as ContextMiner (`http://contextminer.org/`). However, neither of these is able to make information available off-line on a mobile device or offers advanced features such as highly-customisable sources or curation as a service, i. e., curation done *for* the user rather than *by*. This makes WiPo a new and innovative approach to information gathering. In this paper, we will first recapitulate the basics of the innovative WiPo approach and the prototypical implementation which serves as proof of concept and as a vehicle for practical experiments. Subsequently, we demonstrate WiPo's capabilities in the area of film tourism.

2 The WiPo Concept & Architecture

WiPo is a process-based approach to information gathering, Web search, and data curation shown as a high-level Petri Net in Figure 1. It is generic in nature and can be applied to any domain, given domain specific knowledge. The implemented architecture follows the client server principle based on a REST API. Currently, the client is a browser-based GUI, but could be an arbitrary application that can communicate with the server's REST-based Web interface. A comprehensive description of the implementation can be found in [SGH+14].

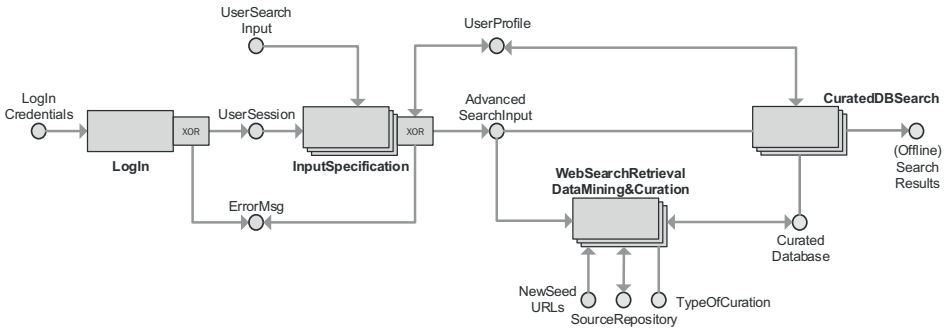


Figure 1: The Overall WiPo Process

First, users log on to the system, resulting in a *UserSession*, and provide *UserSearchInput* which – if valid – can either be profile data that is populated to the stored *UserProfile*, or *AdvancedSearchInput* which must contain a search query and can be enhanced with URLs or private files. *InputSpecification* checks this and creates *AdvancedSearchInput*. This object is then used in two ways. First, it is used to perform *WebSearch, Retrieval, DataMining, & Curation*. In this step, appropriate sources are selected from a *SourceRepository*, to which curators can supply *NewSeedURLs*, and a focused crawl is performed on the selected sources as well as on user-supplied URLs contained in the *AdvancedSearchInput* object. The crawl results undergo data mining and eventually curation (i. e., selection, editing and integration). Eventually, high-quality results will be stored in the *CuratedDatabase*. Secondly, search keywords are read from *AdvancedSearchInput* and are used to query the *CuratedDatabase* in *CuratedDBSearch*. Last, relevant results – considering the user profile – are delivered to the user who then may chose to which to persist offline.

In the prototype, selected URLs are crawled by an Apache Nutch instance. After successful crawling, meta data, i. e., fetch time, last modified time, etc. are determined (stored in the source repository) and content is extracted and passed on to curation – WiPo's distinguishing feature, intended to assure high quality information. In order to achieve this, we suppose that an expert will verify or modify the data mining results prior to their delivery to the user. However, in some cases this expert could also be an algorithm or an externally provisioned service or even a crowd (referred to as *TypeOfCuration*). The curation result will be documented in an indexed repository – the *CuratedDatabase* – implemented using mongoDB and Apache Solr. Thus, WiPo differs from other Web retrieval approaches in that it does not try to solve everything automatically but relies on human curation. A first step towards more automation – since curation is a laborious task – is to allow for automatic propagation of changes to an original source. Therefore, curators have the option to choose what happens, when the source of a curated document changes. The options are *keep*, i. e., text in the curated database remains as is, even if the underlying source changes, *notify*, i. e., curators will see the document again, once it has changed and can then decide what to do, and *auto*, i. e., changes will be propagated unchecked.

The first prototypical implementation which builds the basis for this demo has some minor restrictions. First, it does not yet consider user-supplied documents and as a consequence *InputSpecification, WebSearch, Retrieval,* and *DataMining* are limited. Furthermore, curation is to date restricted to manual curation. Nevertheless this implementation is caplebale of demonstrating the whole process as will be evident in the next section.

3 Use Case & Demo

In this demo[1], we will demonstrate how curators maintain the curated database, as well as how users search for information and how they can tailor the search towards their needs. Therefore, we will present a) the usual work of and b) the data flow between curators and users by the example of cineast tourists.

[1] A video demo can be found at *http://dbis-group.uni-muenster.de/y/wipodemo*.

In the curation demonstration, we will demonstrate the creation of a curated document by New Zealand (NZ) tourism experts. In this context s/he will highlight the use of text segments to combine different sources, multi media content such as images or videos, and map data. As an example, s/he will create a document featuring the Hobbiton movie set containing information regarding, opening hours and entrance fees (from the original Web site) as well as what to particularly look for (sourced from own knowledge or blog entries). This information is enriched with images and video footage as well as general tourism information for the Matamata region (sourced from a variety of Web sources).

Subsequently, the tourist form the introduction with a special interest in Lord of the Rings will search for information on filming locations in New Zealand. This will return for instance the newly created document on the Hobbiton movie set including all additional information such as what else to see in the Matamata region. Given that the tourist knows about the poor mobile Internet coverage in rural New Zealand areas, they wish to persist the information for offline usage with the aid of Evernote. It has to be pointed out that Evernote was a choice of convenience and any other "read-it-later" tool providing an API could have been used. Further, we will go into detail about factors influencing the search result ranking such as previous rating of received documents and the explicit user profiles. To this end, our user will for example change their preference from the North Island to the South Island. As a result, Hobbiton will disappear and new locations will be presented such as the Dry Creek Quarry near Wellington which served as location for Minas Tirith and Helm's Deep.

Following the demonstration of the search using the curated database, we will show how URLs provided by users (e. g. information on a film location tour not yet known to the system such as *http://www.lordoftheringstours.co.nz* are processed. This includes crawling of the supplied URLs, the subsequent curation of the results as well as how the results are eventually presented to users.

References

[Cer10] Sefano Ceri. Chapter 1: Serach Computing. In Sefano Ceri and Marco Brambilla, editors, *Search Computing*, volume 5950 of *Lecture Notes in Computer Science*, pages 3–10. Springer-Verlag Berlin Heidelberg, Berlin and Heidelberg, 2010.

[DSV12] Stuart Dillon, Florian Stahl, and Gottfried Vossen. Towards The Web in Your Pocket: Curated Data as a Service. In *Advanced Methods for Computational Intelligence*, pages 25–34. Springer-Verlag, 2012.

[HSBW13] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013.

[LMH09] C. A. Lee, R. Marciano, and C.-Y. Hou. From harvesting to cultivating. In *Proceedings of the 9th ACM/IEEE-CS joint Conference on Digital Libraries*, page 423, 2009.

[SGH+14] F. Stahl, A. Godde, B. Hagedorn, B. Köpcke, M. Rehberger, and G Vossen. Implementing the WiPo Architecture. In *Proceedings of the EC Web*, München, 2014.

[SHL06] R. Sanderson, J. Harrison, and C. Llewellyn. A curated harvesting approach to establishing a multi-protocol online subject portal: Opening information horizons. In *6th ACM/IEEE-CS Joint Conference on Digital Libraries*, 2006.

MV-IDX: Multi-Version Index in Action

Robert Gottstein #1, Rohit Goyal #1, Ilia Petrov #2, Sergey Hardock #1, Alejandro Buchmann #1

#1Databases and Distributed Systems Group, TU-Darmstadt, Germany,
{gottstein, goyal, hardock, buchmann}@dvs.tu-darmstadt.de

#2 Data Management Lab, Reutlingen University
ilia.petrov@reutlingen-university.de

Abstract: Multi-Versioning DBMS (MV-DBMS) represent a very good match to the properties of Flash storage and the combination of both offers conceptual advantages. Yet, the specifics of indexing in MV-DBMS on Flash have been widely neglected. Although an index in a MV-DBMS references multiple versions of a data item, it is only allowed to return a single (at most one) version of that data item "visible" to the current index operation. Logically separating version visibility checks from the index structure and operations, as in the traditional version-oblivious index, leads to version management overhead: to determine the appropriate version of a data item, the MV-DBMS first fetches all versions that match the search criteria and subsequently discards invisible versions according to the visibility criteria. This involves unnecessary I/Os to fetch tuple versions that do not need to be checked. We propose the idea that version-aware indexing has additional responsibility to recognize different tuple versions of a single data item and to filter invisible tuple versions in order to avoid unnecessary I/Os. In this work we demonstrate an approach called Multi-Version Index (MV-IDX) that allows index-only visibility checks which significantly reduce the amount of I/O as well as the index maintenance overhead. MV-IDX is implemented in the PostgreSQL open source MV-DBMS. We demonstrate that the MV-IDX achieves significantly lower response times and higher transactional throughput on OLTP workloads than the version-oblivious approach. We showcase latency and throughput improvements by utilizing the DBT2 TPC-C benchmarking tool and report saved I/Os. We also showcase how the proposed approach performs better on SSDs.

1 Motivation

MV-DBMS require management of tuple versions of a data item. This has conceptual advantages for Flash [GPB13]: (i) with multi-versioning write operations never block reads, which matches the high read-performance and intrinsic parallelism of Flash; (ii) updates of data items create new tuple versions that are separate physical entities which can be algorithmically utilized to eliminate random writes and ultimately provide write-sequentialisation. However multi-versioning brings up the specific issue of visibility: out of the set of all tuple versions at most one version can be visible to a transaction. Database indices need to handle additional visibility aspects ([JJ07]). [HJS+09] describes a general index structure for MV-DBMS. Traditional (version oblivious) indexing forces the MV-DBMS to determine the visibility after each matching version was fetched (Fig. 1), since

such approaches store visibility information physically on each version. In addition the index needs to contain records referencing all existing tuple versions. Visibility can therefore only be determined with additional I/O access [GGH+14]. Since version management is handled by the MV-DBMS concurrency control, the index cannot optimize accesses to invisible tuple versions.

In this demonstration we present the Multi-Version Index (MV-IDX), a version-aware index structure that is capable of answering visibility decisions efficiently in-memory, solely by accessing the index and logically working on two dimensions: the indexed attribute and the data item. Independently of the underlying physical storage layout, a tuple's visibility can be determined before it is fetched from disc, thus reducing the overall I/O to the storage device. Furhtermore index management overhead is avoided on updates that do not change the search-key value. The MV-IDX is implemented into the PostgreSQL MV-DBMS, using *Snapshot Isolation* concurrency control with the *first-updater-wins* rule. We showcase latency and throughput improvements using OLTP workload generated by the TPC-C benchmark and compare it to the unaltered conventional indexing within PostgreSQL. We show that the MV-IDX achieves significantly lower response times as well as higher throughput on OLTP workload (Fig. 2). In addition we present the amount of saved I/O accesses by the MV-IDX. The audience can vary the workload using the TPC-C/DBT2 parameters. In addition we offer specific microbenchmarks to stress special workloads on the MV-IDX.

2 The Multi-Version Index

Figure 1 depicts the basic structures of the demonstrated MV-IDX algorithm. The MV-IDX uses a virtual identifier (VID) that uniquely identifies a data item and all indexed tuple versions belonging to it. Instead of a tuple identifier (TID) the VID is stored in the index: the typical MV-IDX index record comprises a *Key-VID* pair, where the traditional index records comprise *Key-TID* pairs. A single data item is therefore identified by a VID and each tuple version by its TID.

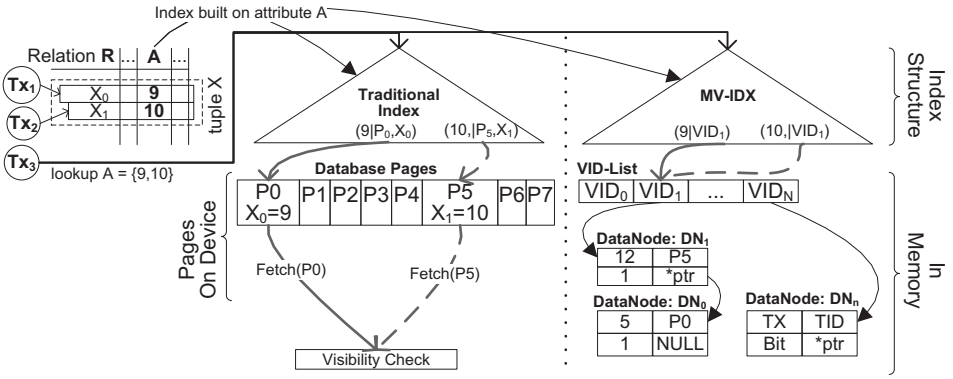


Figure 1: MV-IDX Example and Major Data Structures

Data Structures The MV-IDX uses two in-memory data structures (Fig. 1):

(i) The *VID-List* mapping structure that stores a pointer for each VID to a single *data node*. The pointer always points to the data node that describes the newest (most recent) tuple version. (ii) One *Data Node* per tuple version, comprising: *a) the transaction timestamp* (TX) and *b) the TID* of the tuple version; *c)* a *Bit* that indicates whether the timestamp (a) denotes the commit time of the tuple version or its insertion time (e.g., the inserting transaction is still active) and *d) a pointer* $*ptr$: If a potentially visible tuple version directly preceding the most recent one exists, the pointer refers to the *data node* that stores information about it; otherwise a *NULL* value is stored.

Size and Management The size of the in-memory data structures is deterministic and reasonable, the $VID - List$ has to store n data node pointer, where n denotes the amount of date items in the relation. The size of each node sums up to 136bit: the transaction ID (TX, 32bit), tuple ID (TID, 64bit), status bit (Bit) and the pointer ($*ptr$, 32bit). Each entry in the $VID - List$ uses 32bit. Data nodes that describe tuple versions which are not visible to any running transaction any more can be garbage collected.

Algorithm: Fig. 1 depicts an example of the MV-IDX to clarify its principle, details are provided in [GGH+14]. Relation R has an index on attribute A. Consider transaction TX_1 that *inserts* a new data item X and thus creates the first tuple version X_0. X receives the unique VID_1, identical among all tuple versions X_i of X. A Data Node DN_0 is created, containing the VID (VID_1), TID of the new tuple version ($P0$) and the transaction id of the creating transaction TX_1 (TX) are set. The bit is set to 0 to indicate that the tuple version is not yet committed; **ptr* is set to *NULL* since a preceding version does not exist. A pointer directing to DN_0 is stored in *VID-List*. The *Key-VID* pair is inserted at the appropriate position within the MV-IDX.

If a concurrently running transaction TX_{read-1} (not depicted) performs an index lookup that matches the key value of the new data item, the VID is retrieved and using the $VID - List$, the data node DN_0 is accessed. The status Bit in DN_0 is reset, indicating that the tuple version is not yet committed (TX shows that the transaction is concurrently running) - the visibility check therefore returns *Not-Visible*. TX_1 commits and finishes at time 5, the bit in DN_0 is set to 1 and the TX is set to 5. TX_{read-2} (not depicted) starts and requests to read X. The access on the MV-IDX leads to DN_0. The status bit indicates that the version is committed and visible since TX_{read-2} is greater than TXin DN_0. Updates proceed analogously to inserts. A new data node DN_1 is created (depicted in Fig. 1 as TX_2) that describes the new tuple version (VID, TID and TX fields are set accordingly). The pointer in *VID-List* is set to refer to DN_1 and the **ptr* on DN_1 refers to DN_0. Older transactions follow the $*ptr$ reference to DN_0.

So far TX_1 inserted data item X in version X_0, committed at transactional time 5, TX_2 creates the update X_1 and commits at transactional time 12. The value of attribute A changed and the index contains both versions of X. Note: If the indexed key value has not been changed nothing has to be updated or stored additionally in the MV-IDX - the *VID-List* points to the new data node.

TX_3 executes a lookup of all data items with values $A = \{9, 10\}$. The traditional index fetches each version, contained in two different pages from disc, resulting in 2 disc I/Os.

Both versions are checked for visibility - X_0 is discarded. The MV-IDX retrieves the VID and finds the only visible tuple version X_1 without the need for any disc I/O - independent of the underlying storage device (SSD, HDD). The visible tuple version can be fetched afterwards. The approach is also applicable to multicolumn searchkeys.

3 Demo Scenario

In our first demo scenario we showcase the MV-IDX implementation in PostgreSQL (Version 9.3.4). We execute testruns of the TPC-C benchmark using the open source DBT2 implementation. The results are compared to the unaltered PostgreSQL DBMS. The audience can alter the duration of the benchmark as well as the amount of warehouses. Varying these parameters shows the influence of the amount of tuple versions per data item, results are depicted in Fig. 2. The MV-IDX improves on the native implementation especially on longer testruns where the amount of tuple versions increases.

In our second demo scenario we execute microbenchmarks that directly point out the influence of a high amount of tuple versions per data item. Long running transactions are mixed with short running transactions in order to stress the management functions of the in-memory data structures to demonstrate their efficiency in access speed and memory consumption. In addition we sum up the amount of saved I/O accesses by the MV-IDX.

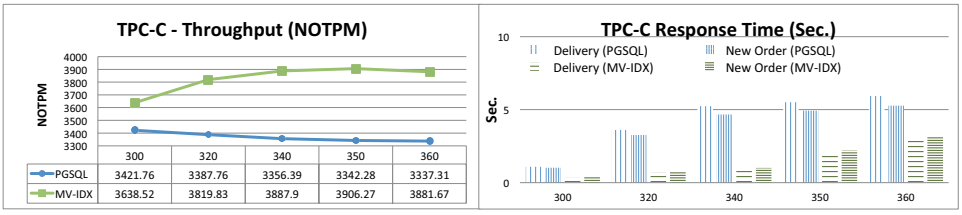


Figure 2: TPC-C on SSD: Throughput - Latency - 2h Runtime - 15 Clients - 150 to 360 Warehouses

Acknowledgments This work was supported by the DFG (Deutsche Forschungsgemeinschaft) project "Flashy-DB".

References

[GGH+14] Robert Gottstein, Rohit Goyal, Sergej Hardock, Ilia Petrov, and Alejandro Buchmann. MV-IDX: indexing in multi-version databases. In *IDEAS'14*. ACM, 2014.

[GPB13] Robert Gottstein, Ilia Petrov, and Alejandro Buchmann. Append storage in multi-version databases on flash. In *Big Data*. Springer Berlin Heidelberg, 2013.

[HJS+09] Tuukka Haapasalo, Ibrahim Jaluta, Bernhard Seeger, Seppo Sippu, and Eljas Soisalon-Soininen. Transactions on the multiversion B+-tree. In *Proc.*, EDBT '09, 2009.

[JJ07] Khaled Jouini and Geneviève Jomier. Indexing multiversion DBs. In *Proc CIKM*, 2007.

Online Bit Flip Detection for In-Memory B-Trees Live!

Till Kolditz, Benjamin Schlegel*, Dirk Habich, Wolfgang Lehner

Database Technology Group
Technische Universitt Dresden
01062 Dresden, Germany
{first.last}@tu-dresden.de

Abstract: Hardware vendors constantly decrease the feature sizes of integrated circuits to obtain higher performance and energy efficiency. As a side-effect, integrated circuits – like CPUs and main memory – become more and more vulnerable to external influences and thus unreliable, which results in increasing numbers of (multi-) bit flips. From a database perspective bit flip errors in main memory will become a major challenge for modern in-memory database systems, which keep all their enterprise data in volatile, unreliable main memory. Existing hardware error control techniques like ECC-DRAM are able to detect and correct memory errors, but their detection and correction capabilities are limited and come along with several downsides. To underline this we heat up RAM live on-site to show possible error rates of future hardware. We previously presented various techniques for the B-Tree – as a wide-spread index structure – for online error detection and thus increase its overall reliability [Kea14b]. We also show live performance comparisons in terms of throughput and error detection rates between several bit flip detecting B-Tree variants. By that, we demonstrate the tradeoff between detection accuracy and index throughput. Furthermore, we show annotated structural information about the trees like corrupted nodes and inaccessible sub-trees.

1 Introduction

For the last 30 years disk-centric database systems have been state-of-the art, but in the last couple of years this approach has dramatically changed due to several reasons. Especially the significant developments in the hardware sector are the major driver for that change. Due to these hardware developments, servers with 2 TiB of main memory are available for a reasonable price. To efficiently leverage the provided main memory capacities, the database system architecture trend shifted from a disk-centric to a main memory-centric approach, where the entire data pool is kept completely in main memory. The performance gain is massive because database operations are able to benefit from its higher bandwidth and lower latency. However, due to the constant decrease of transistors' feature sizes cosmic rays [BSS08], heat dissipation or electric interaction [Kea14a] have increasing impacts on transistor state stability, which lead to arbitrary (possibly multi-) bit flips (MBFs). Studies show that current hardware is already subject to many bit flips, and even to a substantial portion of multi bit flips. As a first contribution, we will heat up RAM live on-site to show

*currently working at Oracle Labs

behavior of current hardware at extreme operational temperatures (about $130°C$), which gives an outlook on future hardware being error-prone at much lower temperatures. Applications must be hardened to detect and handle MBFs in memory, especially in the case of (main memory) databases, because users or customers typically regard data stability as an essential property of database systems. Data stability is, however, orthogonal to, e.g., the ACID properties concerning transactional behavior of databases. Current techniques like error correction code (ECC) RAM on the one hand introduce higher latencies and monetary costs, while on the other hand they will not suffice to deal with increasing MBF rates. Furthermore, such general purpose techniques cannot leverage application-specific knowledge about applied data structures as, e.g., in DBMSs. Therefore, we proposed several light-to-heavy-weight error (bit flip) detection adaptations to the still wide-spread B-tree [Kea14b], for whose many variants like B+ trees or B*-trees these error detection mechanisms can be adopted. The bit flip detection is integrated in an online fashion during tree traversal, whereas there is no error correction published yet. As a second contribution, in a live demo we visualize the benefits and tradeoffs of our proposed error-detecting B-Tree variants, especially the tradeoff between throughput and protection against main memory MBFs. Due to the size restrictions, please refer to [Kea14b] for any related work.

2 Online Bit Flip Detection for In-Memory B-Trees

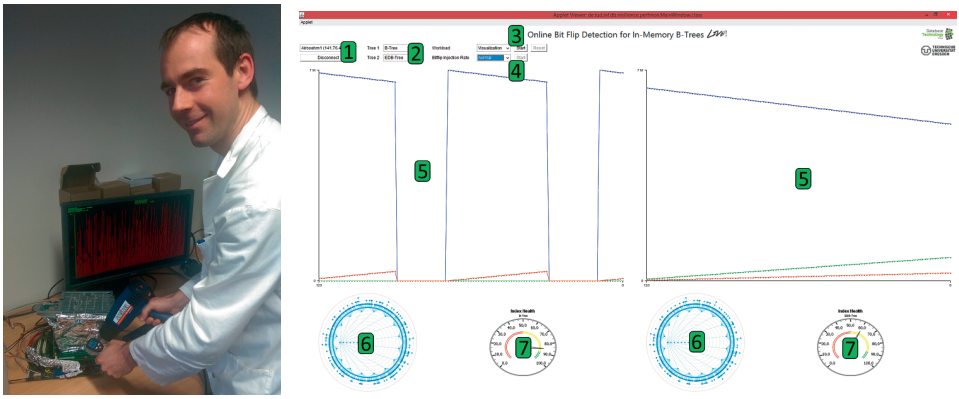
As a first step towards database systems being resilient against arbitrary main memory MBFs we introduce our adaptations of the widely used index structure B-tree [Kea14b].

TMR B-Tree: Triple Modular Redundancy (TMR) is a common technique for tolerating arbitrary data corruption, where three copies – replicas – of all data are maintained and algorithms are executed on all three copies. The results are compared and finally a majority voting should lead to the correct result. Execution time and required amount of memory are tripled, which might not always be acceptable. We accordingly also demonstrate a TMR B-tree variant.

EDB-tree: Our first error-detecting B-tree adds pointer sanity checks. Therefore, we exploit the large virtual address space and allocate the whole tree in one contiguous memory area. By that, pointers pointing outwards of this region can be identified. Also, back-links are added and validated against the originating node's address during tree traversal (parent-child relation). Finally, it is checked whether the address which is pointed at is aligned to the node size. These checks detect most if not all pointer corruptions.

EDB-Tree PB: Our second adaptation sets parity bits for each and every node member, by which also corruption in nodes' meta data, keys and values can be partially detected. Each node member's value domain is stripped by the most significant bit (MSB), which in return is set to obtain an even parity. During tree traversal whenever a node member is accessed the first time, its parity is first checked for evenness. Simple parities only detect any odd-numbered bit flip in a data word.

EDB-Tree CS: In our third variant, for detecting more bit flips we employ XOR checksums to groups of node members. On the one hand these can be computed easily and quickly by XOR-ing the concerning data elements, while on the other hand they will detect almost any bit flips. The exception is when exactly an even number of bit flips occurs at the very



(a) Heating RAM

(b) Live application GUI

Figure 1: Setup for the demonstration.

same bit position in an even number of data objects, so that these even out each other. We add four checksums to each node: one for a node's meta data, all keys, all values and all pointers, respectively. During tree traversal, a checksum is validated before one of the according node members (over which it was computed) is accessed.

3 Demo Description

With this demo we contribute two things. First, we show a video on how we heat up RAM and measure MBFs in main memory. Second, we visualize the benefits and tradeoffs of our proposed error-detecting B-Tree variants, especially the tradeoff between throughput and protection against main memory MBFs.

RAM on Fire Live on-site we heat up RAM to demonstrate possible error rates of future hardware (cf. Figure 1a). An application checks a portion of memory for MBFs, while virtually setting the RAM on fire. We therefore use a heat gun and raise the memory chips' surface temperature to $100 \ldots 130°C$. A live diagram displays the bit flip distribution across the checked memory range.

Error Detecting B-Trees Live! For visualizing different aspects of our proposed error detecting B-trees we will present an application shown in Figure 1b. The demo setup and details highlighted in the figure as well as the demo experience are described in the following. For the demo an Intel server-grade machine will be available running an Intel i7-3960X 12-core CPU (including hyper-threading) and 32 GiB RAM. The GUI interaction starts with the user connecting to the machine (1 in Figure 1b). The available B-tree variants can then be selected from the according combo boxes labeled "Tree 1" and "Tree 2" (2 in Figure 1b). By that, the user can choose which B-tree variants she or he wants to compare against each other. Next, the user selects a workload (3 in Figure 1b) depending on which particular graphs are shown (see below). Afterwards, the user may start, pause and resume the workload and reset the trees, i.e. have them rebuilt (also 3 in Fig-

ure 1b). The user may choose between 3 different workloads: performance, visualization, and manual, as presented below.

Data corruption: Additionally, the user may select a bit flip injection rate to control how fast data is corrupted (4 in Figure 1b). To simulate future error rates, bit flips are synthetically induced into arbitrary data words in the indexes. While running a workload, at the selected bit flip rate a random 8-byte word is XORed against a bit pattern of 1 - 5 ones to simulate accordingly many bit flips.

Performance Workload: The performance workload compares the throughput between two tree variants (indexes). It runs constant point queries on the selected indexes while corrupting arbitrary nodes randomly. This workload only displays the 2 line graphs (5 in Figure 1b). It is intended to show the performance impact of the different error detection variants using 3 different lines: Blue represents the throughput of successful queries without any errors. Red represents undetected errors (cf. [Kea14b]), which means false positives, false negatives or segmentation faults, while green stands for the amount of errors detected by the according tree variant. As can be seen in Figure 1b trees might have to be rebuilt when encountering a segmentation fault – segmentation faults lead to program termination and are much harder to recover from in contrast to false positives or negatives. Therefore a complete rebuild of the tree is required.

Visualization Workload: The visualization workload additionally shows a graphical representation of the trees themselves – as multi-layered circled graphs – and health indicators – in the form of dials – (6 in Figure 1b). This allows the user to see the current structure of the tree and its evolving over time, i.e. which parts of the tree are corrupted, and which key ranges or nodes are not accessible anymore. Thereby, blue dots represent accessible nodes, black dots stand for corrupted nodes, and red nodes show inaccessible nodes. The health indicators represent the amount of queries which are successful.

Manual Workload: The manual workload also displays all graphs, and there is also a constant load of point queries. However, the user may directly select a node which she or he wants to corrupt. The corruption is done randomly and visual feedback immediately shows what happens due to the corruption. For instance, when a child pointer was corrupted, the according sub-tree is marked red.

Acknowledgments

This work has been supported by the state of Saxony under grant of ESF 100111037 (SREX) and the German Research Foundation within cfAED cluster of excellence.

References

[BSS08] L. Borucki, G. Schindlbeck, and C. Slayman. Comparison of accelerated DRAM soft error rates measured at component and system level. IRPS, 2008.

[Kea14a] Yoongu Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. ISCA. IEEE Press, 2014.

[Kea14b] Till Kolditz et al. Online Bit Flip Detection for In-memory B-trees on Unreliable Hardware. DaMoN. ACM, 2014.

RecoLeta: A Recommender System for Events for Personalised E-Mail Campaigns

Frank Eichinger, Immanuel Wietreich

CTS Eventim AG & Co. KGaA, Information Science, Hamburg, Germany
{frank.eichinger, immanuel.wietreich}@eventim.de

Abstract: We demonstrate the RecoLeta system for event recommendations. It combines two different recommender approaches: one novel approach dedicated to music concert events and one state-of-the-art approach. We also present our big-data architecture for e-mail delivery and recommendation calculation in an in-memory database.

1 Introduction

In e-commerce, e-mail marketing is a key tool to attract customers to web pages and to stimulate orders. Apart from a few data-driven companies and early adopters, many companies send out static newsletters, e.g., millions of customers receive the same content. Typically, the content is selected manually by editorial staff and focuses on top-seller products. The reason is that the content should match the interest of a potentially large share of the customers. As this content is popular and frequently well-known, the likelihood that customers buy the respective products is relatively high. The drawback of this is that the mid and long tail of products [LRU14], i.e., the non-top-seller products, will not be advertised and will ultimately be bought even less frequently. Personalised recommendations have the potential to feature the mid and long tail and to recommend only products to customers they are likely to be interested in.

Even if there is a huge amount of literature on recommender systems (see, e.g., [RRSK11]) including dedicated recommender systems for music (see, e.g., [KDK11]), there is few literature for the recommendation of events such as music concerts, sport and culture events. Such events are however different than other products like books or CDs, as they have an associated date and location (the recommendation of historic

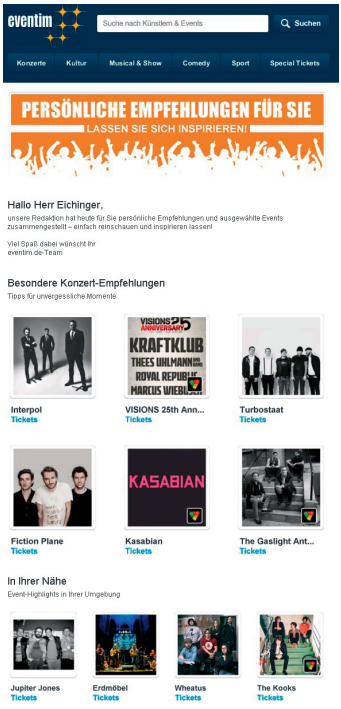


Figure 1: A newsletter containing personalised event recommendations. Contents are dynamically loaded when the e-mail is opened.

events is not relevant; event venues with a large distance to the users might not always be promising). Furthermore, there are artists doing tours with several associated events which should be taken into account when recommending events to customers.

In this paper, we present the RecoLeta system for event recommendations, which has a focus on music concerts. This system combines a content-based approach [LRU14] (utilising music genre information) for recommending events of artists the users probably like with a state-of-the-art item-to-item collaborative-filtering approach [LSY03]. Furthermore, it incorporates event-specific information such as dates and locations. This approach ensures that customers receive recommendations from music artists which may be new to the customers but still relevant, and that the customers may receive other events than concerts as well. It also covers customers who are not interested in concerts. As input for the recommendations, we use a variety of data ranging from orders to customer demographics. Besides the recommender system, we describe our big-data infrastructure for warehousing, recommendation calculation, campaign management and e-mail (content) delivery.

The RecoLeta system and surrounding infrastructure is a live system which we are currently testing in a continuous-improvement process. Specifically, we send newsletters to millions of real customers (see Figure 1 for an example). First results indicate that dynamic recommendations increase the number of clicks, orders, sold tickets and revenue compared to a static variant with editorial content in mid and long-tail promotions.

2 The RecoLeta System for Calculating Event Recommendations

Content-Based Approach for Music Concerts. In order to recommend music concerts to customers, we follow a content-based approach which compares music profiles from customers (building on their past interests) to those from artists. This allows us to recommend artists to customers they are likely to be interested in but they might not know so far. Our intention is to inspire the customer and to feature concerts from the mid and long tail.

In order to calculate music similarities, we rely on genre information from the project `www.discogs.com`. This wiki-like web page contains user-generated information for more than five million albums. Each album is assigned with one or more of 15 'genres' and 382 'styles' (sub-genres). Based on this information, we have derived artist profiles. These are vectors where the elements describe the degree to which an artist belongs to a certain music genre. As an example, the artist profile for 'The Rolling Stones' is a vector with value 89% for 'rock', 4% for 'blues', few further fields with small values and many fields with value 0%. Using distance functions [LRU14], we can calculate similarities between two artists. For example, the similarity based on Manhattan distance of 'The Rolling Stones' and 'The Who' is 81%, while the similarity of 'The Rolling Stones' and 'Helene Fischer' is 3%. Likewise, we calculate user profiles as vectors containing the aggregated values of all artist profiles (vectors) from artists the customers have showed interest in. Having artist profiles and customer profiles, we calculate raw recommendations as similarities from all customers to all artists we have concert tickets available for. As long as we have an artist profile, this approach does not suffer from any cold-start problems.

Item-to-Item Collaborative-Filtering Approach. While the content-based approach allows for recommendations of music concerts, item-to-item collaborative filtering is a general approach for recommendations [LSY03]. This allows for both music and non-music recommendations – for customers having shown interest in music and/or other types of events alike. It calculates a similarity matrix of items. On the input side, the items are events (or tours or artists) customers have bought tickets from. On the output side, the items are events (or tours or artists) for which tickets are available. The cells contain the cosine similarity [DK04, LRU14, LSY03] of both events. Roughly speaking, values are high when the number of customers who have bought tickets from the input and the output side is high. We use all events a customer has been interested in as an input to calculate raw recommendations, using matrix-vector multiplication [DK04]. There are usually no cold-start problems, as frequently many tickets are sold on the fist day they are available.

Selection of Recommendations for a Customer. When both approaches have calculated a raw set of recommendations for each customer, we merge these sets using information on music-data availability per customer for steering the weight of each approach. This results in a base set of recommendations for each customer. We then filter this set according to several categories we want to feature in a campaign. The example in Figure 1 contains one category for concert recommendations which have the highest similarity values and one category which contains possibly all kinds of events ordered by similarity and physical distance between the customer's place of residence and the event venue.

3 Recommendation and Content Delivery Architecture

Figure 2 describes our big-data architecture for recommendation calculation and content delivery. When a customer orders event tickets in the web shop (1), order information is saved in the web-shop database (2). Using ETL processes, we pseudonymise and copy this data and data from other sources to our data warehouse (3). The warehouse is an EXASolution in-memory column-store database cluster, where we also run all recommendation-calculation processes (4; see Section 2) on a daily basis. We also transfer a subset of the data to our campaign-management system DynaCampaign (6), which writes campaign data back to the warehouse (6). This system manages target and control groups and exports e-mail recipients along with salutations to an e-mail service (8). This service sends out e-mails to the customers (9). These e-mails contain only place holders (identified by '?' in the figure) with underlying personalised links rather than actual content like event recommendations. The customer's e-mail client then follows these links (10) which access our e-mail content delivery system. This system has received pre-calculated recommendations for all users from the warehouse (5) in a pseudonymised way. After a ticket-availability check, it redirects the links to the actual image and link URLs of recommended events (11). The e-mail client then accesses the actual images representing teasers for recommended events from a content delivery network (CDN; 12-13; in the figure, five place holders have already been replaced with images). If the events are interesting to the user, she or he clicks them (14) and reaches the home page where the event tickets can be ordered (1).

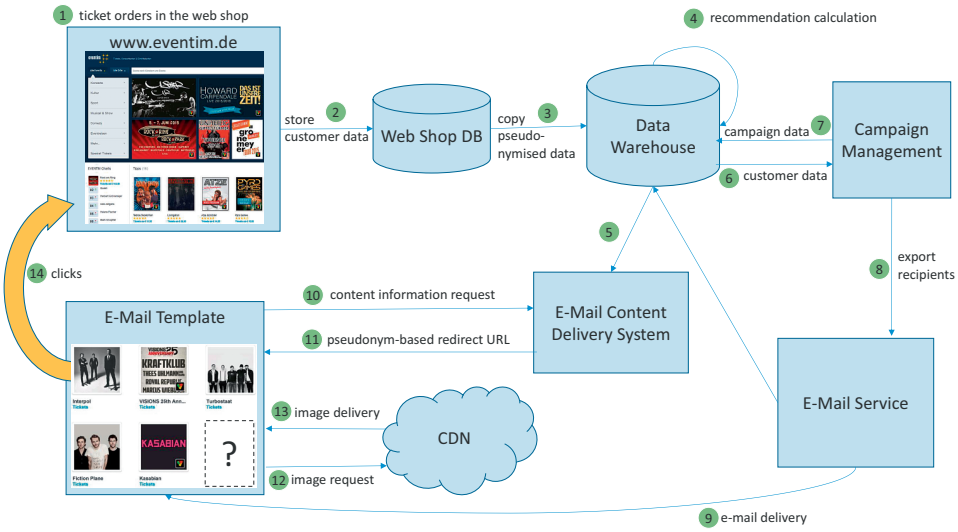


Figure 2: Big-data architecture for recommendation calculation and e-mail delivery.

4 Demonstration at the Conference

At the conference, we will demonstrate the RecoLeta recommender system. Conference attendees will have the possibility to receive personalised event recommendations from our live system by entering their preferences into an iPad. As we are currently performing a live test within our direct marketing with millions of real customers over several months, we will also report on the results and experiences of this test at the conference.

References

[DK04] Mukund Deshpande and George Karypis. Item-based Top-N Recommendation Algorithms. *ACM Transactions on Information Systems*, 22(1):143–177, 2004.

[KDK11] Noam Koenigstein, Gideon Dror, and Yehuda Koren. Yahoo! Music Recommendations: Modeling Music Ratings with Temporal Dynamics and Item Taxonomy. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2011.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. amazon.com Recommendations: Item-to-Itemtem Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[RRSK11] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer, 2011.

Sequential Pattern Mining of Multimodal Streams in the Humanities

Marwan Hassani [•] Christian Beecks [•] Daniel Töws [•]
Tatiana Serbina [○] Max Haberstroh [§] Paula Niemietz [○]
Sabina Jeschke [§] Stella Neumann [○] Thomas Seidl [•]
[•]Data Management and Data Exploration Group
{hassani, beecks, toews, seidl}@cs.rwth-aachen.de
[○]Group of English Studies [§]IMA/ZLW & IfU
RWTH Aachen University, Germany

Abstract: Research in the humanities is increasingly attracted by data mining and data management techniques in order to efficiently deal with complex scientific corpora. Particularly, the exploration of hidden patterns within different types of data streams arising from psycholinguistic experiments is of growing interest in the area of translation process research. In order to support psycholinguistic experts in quantitatively discovering the non-self-explanatory behavior of the data, we propose the *e-cosmos miner* framework for mining, generating and visualizing sequential patterns hidden within multimodal streaming data. The introduced $\mathcal{M}$SS-BE algorithm, based on the *PrefixSpan* method, searches for sequential patterns within multiple streaming inputs arriving from eye tracking and keystroke logging data recorded during translation tasks. The *e-cosmos miner* enables psycholinguistic experts to select different sequential patterns as they appear in the translation process, compare the evolving changes of their statistics during the process and track their occurrences within a special simulator.

1 Introduction

In the area of translation process research, psycholinguistic experiments are conducted to draw conclusions on the cognitive processing that takes place during translation tasks. The cognitive dimension, operationalized in terms of gaze- and keystroke-related statistics, can be used to explain, among other things, linguistic features of the translated language identified with the help of corpus studies (analysis of large and representative collections of translated and non-translated texts). An example of a research question is the analysis of grammatical complexity both in the final translation product and in the intermediate versions of the translation process: the psycholinguistic perspective can extend the corpus-based analysis of the phenomenon by answering such questions as whether grammatically more complex items require more cognitive processing and what translation strategies are employed to produce a more or less complex linguistic structure [APN$^+$10]. One of the aims of the *e-cosmos* project is to treat the process data, i.e. the unfolding translations, as another type of corpus . This corpus will be queried for both behavioural and linguistic information, which researchers can analyze quantitatively while still taking into account

fine-grained details. The application of data-mining techniques allows us to investigate the data in a bottom-up manner; this approach yields multimodal patterns which may otherwise be overlooked yet contribute to the explanatory potential of the data. Within the *e-cosmos* project, we are asked to analyze the multimodal data collected during a psycholinguistic translation experiment. Sixteen participants (eight professional translators and eight domain specialists, i.e. PhD students of physics) were asked to translate an abridged popular-scientific text from English into their native language (German). Their keystrokes, mouse movements and pauses in between were recorded . Additionally, eye-tracking data during this translation task was gathered using the remote eye tracker *Tobii* 2150. It contains information on individual gaze points that have been collected with a 50 Hz sampling rate (50 gaze points per second) for each translation session. The remainder of this paper is organized as follows: Section 2 introduces the $\mathcal{M}$SS-BE algorithm used mainly by our demo, then in Section 3, we explain the analysis and interpretation of patterns using the proposed *e-cosmos miner* demonstrator as well as the demo plan.

2 The $\mathcal{M}$SS-BE Algorithm

The SS-BE algorithm was suggested in [MDH08] to find sequential patterns in a single data stream. This algorithm breaks a stream into batches and performs a single scan over every batch using the *PrefixSpan* algorithm [PHMA$^+$04]. It promises a bounded error by guaranteeing that all true patterns in any batch are output at the end of that batch. In [HS11], some ideas were discussed on using the SS-BE algorithm for mining multiple streams with the aim of gaining a health context prediction. The SS-BE algorithm has a nice, stream-friendly, feature of pruning and updating the $T0$ lexicographic tree containing the PrefixSpan output. However, it has two main problems which make it insufficient for dealing with the multimodal streams in our humanities scenario: (1) It is unable to discover patterns within multiple streams, (2) Its batch-based method might results in losing all interesting sequential patterns whose sub-sequences appear on two batches. We propose in this demo the $\mathcal{M}$SS-BE algorithm (Multiple Stream Sequence miner using Bounded Error) in order to overcome the aforementioned drawbacks. $\mathcal{M}$SS-BE seeks sequential patterns that consist of consecutive patterns belonging to multiple streams The multiple stream mining concept is mainly based on the a-priori observation. When using an equal or larger threshold value, any sequence of consecutive patterns that belong to multiple streams is *frequent* if and only if these consecutive patterns were *already frequent*. The $\mathcal{M}$SS-BE algorithm we suggest in this demo can be summarized in the following main steps: **1.** Generate all sequential patterns appearing in every stream for finding *intra-stream* sequential patterns. Input are patterns that consist of single-item, timestamped sequences. These timestamps for each batch are added to the end of that sequence in $\mathcal{M}T0$ tree structure (a modified structure of [MDH08]). An example of the output is detailed in Fig. 1 (Top), by assuming that a *window width* = 40 and *min_supp*= $\frac{1}{40}$ (i.e. a sequence is frequent if it appears at least once). Paths consisting of more than one node are representing *consecutive* sequences frequently seen. **2.** To find the *inter-stream* sequential patterns, for every window, a table similar to that in Fig. 1 (Bottom) is built. Using a *min_supp*= $\frac{2}{40}$ on the inter-stream level, the resulted inter-stream sequential pattern from our example is: $\langle LE, lo\rangle$:2, which represents a two-times appearance of typing a letter

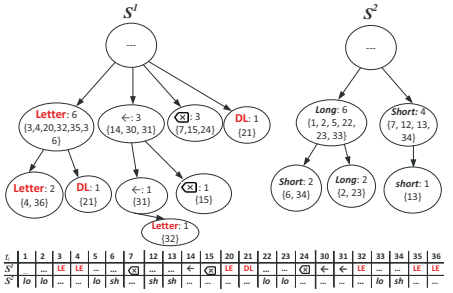
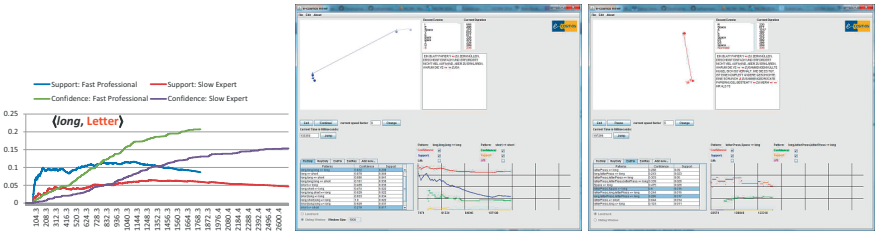


Figure 1: **Top:** An example of the state of the $\mathcal{MT}0$ tree for streams S^1: *KeyStrokes* and S^2: *Eye Gazes* showing in each node: [*sequential pattern*: *count* {*ending timestamps*}]. **Bottom:** The timing sequence of sequential patterns for generating the *inter-stream* rules based on consecutive sequential patterns that belong to different streams. *LE= letter, DL= delete, sh= short gaze, lo= long gaze*.

followed by a long gaze in the current window. **3.** Continuously maintain the nodes in the $\mathcal{MT}0$ tree as the window slides (no batch processing). Delete an item if all its ending timestamps are older than the current starting timestamp of the window. Decrement its count if one of its ending timestamps older than the window starting timestamp. Insert newly seen items in their correct available/new place in the tree and increment/initiate the count. Delete the nodes that represent infrequent items.

3 Analysis of Patterns With the *e-cosmos Miner* Demo



(a) (b) (c)

Figure 2: (a) A Comparison between a fast professional translator and a slow domain expert (physicist) translator over the time (in Seconds) obtained by the *e-cosmos miner* for the pattern: $\langle long, Letter \rangle$. Screen shots of the *e-cosmos miner* comparing: (a) Two intra-stream rules with the sliding window concept, (b) Two multimodal rules with the landmark window concept.

The *e-cosmos miner* reads the eye tracking and keystroke logging in a streaming way to observe the changes of the patterns as the translation session evolves. The *e-cosmos miner* currently identifies *intra-stream* (single) and *inter-stream* (multimodal) behavioral sequential patterns consisting of the following elements: short or long fixations as well as different types of characters typed by the participants (cf. Fig. 1). Previous research has established that average fixation duration during silent reading (for comprehension) is approximately 200 milliseconds [JJ08]. A distinction for the characters was done to differentiate normal from other control characters as: Delete, Backspace, Arrows, left mouse click, etc. (cf. Fig. 1).

An example of an identified multimodal pattern is a long fixation followed by a letter.

Since long fixations are typically interpreted as one of the concrete indicators for increased cognitive demand , this pattern might appear during processing of the linguistic unit that is produced after a long fixation. In Fig. 2(a), we picked a professional translator who required the least amount of time for translation to compare them to the domain specialist characterized by the longest translation session. Some statistics of the rule: $long \Rightarrow Letter$ which is implied by the pattern $\langle long, Letter \rangle$ are depicted in Fig. 2(a). The changes of the confidence and the the support values over time are different for the two participants. The figure shows a first indication of potential differences between the two groups. Moreover, these first patterns contribute both to an understanding of how to appraise patterns in general, and to increasing sophistication in the types of patterns suggested by the *e-cosmos miner*. As can be seen in Fig. 2, an external file containing the timestamped gazes and keystrokes can be read within the *e-cosmos miner*. By selecting the size of the sliding window, or selecting the landmark window, clicking the *start* button (which will flip to be an *end* button) will start the simulation. The recent eye gazes, their duration, their location and movements are simulated in the upper part of the GUI together with the current key strokes and the resulting text. All emerging patterns are depicted in the bottom and the statistics like: (confidence, support and lift) are plotted against the evolution of the real time in the bottom right corner. The psycholinguistic expert can filter the observed patterns according to: one-dimensional vs. multimodal (cf. Fig. 2(b) vs. Fig. 2(c)), or according to stream of the ending item of the pattern. Additionally, the simulation and generation of patterns can jump to any time of the session, and the simulation speed can be increased or decreased. BTW participants can observe the effect of varying different streaming parameters over the process of building the $\mathcal{MT}0$ tree and the intermediate top-k patterns of a single and of multiple translators and match those to the current translation step. The *e-cosmos miner* can be downloaded from our website: `http://dme.rwth-aachen.de/en/research/projects/e-cosmos/ecosmos_miner`.

Acknowledgments

Funded by the Excellence Initiative of the German federal and state governments.

References

[APN+10] F. Alves, A. Pagano, S. Neumann, E. Steiner, and S. Hansen-Schirra. Units of translation and grammatical shifts: towards an integration of product- and process-based research in translation. In *Translation and Cognition*, pages 109 –142. 2010.

[HS11] Marwan Hassani and Thomas Seidl. Towards a Mobile Health Context Prediction: Sequential Pattern Mining in Multiple Streams. In *MDM '11*, pages 55–57, 2011.

[JJ08] A. L. Jakobsen and K.T.H Jensen. Eye movement behaviour across four different types of reading task. In *Looking at eyes: eye-tracking studies of reading and translation processing*, pages 103 –124. 2008.

[MDH08] L.F. Mendes, Bolin Ding, and Jiawei Han. Stream Sequential Pattern Mining with Precise Error Bounds. In *ICDM '08*, pages 941 –946, 2008.

[PHMA+04] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Trans. on KDE*, 16:1424–1440, 2004.

SQL-Grundlagen spielend lernen mit dem Text-Adventure SQL Island

Johannes Schildgen, Stefan Deßloch

Heterogeneous Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{schildgen,dessloch}@cs.uni-kl.de

Abstract: Wir präsentieren SQL Island, ein neuartiges browserbasiertes Lernspiel, welches auf dem Konzept der Text-Adventures basiert. Nach einem Flugzeugabsturz landet die Spielfigur auf einer Insel. Man redet mit Bewohnern, sammelt Gegenstände und käpft gegen Bösewichte. Die Besonderheit bei diesem Spiel ist jedoch, dass der Spieler seine Figur lediglich mittels SQL-Befehlen steuert. Alle nötigen Befehle werden zunächst präsentiert, sodass keine Vorerfahrung notwendig ist. Nach etwa einer Stunde Spielzeit beherrscht der Spieler SELECT-, UPDATE- und DELETE-Anfragen sowie Gruppierungen, Aggregationsfunktionen und Joins. Das Spiel kann online auf `http://www.sql-island.de` kostenlos und ohne Registrierung gespielt werden.

1 Motivation

Die Technische Universität Kaiserslautern bietet mehrmals im Jahr Workshoptage[1] für Schülerinnen und Schüler an, um diesen Einblicke in die Universität und in das Studium zu gewähren. Ein wesentlicher Bestandteil dieser Veranstaltungen sind Workshops, bei denen selbst etwas ausprobiert werden kann. Beliebt sind Brückenbauworkshops, das Programmieren von Lego-Robotern sowie - vor allem beim Schülerinnentag[2] - das Mischen eigener Parfüms. Der bis vor wenigen Jahren angebotene Datenbank-Workshop, in dem die Schüler Bücher über eine SQL-Kommandozeile in einen Webshop einfügen konnten, stieß dabei jedoch auf nur weniger großes Interesse. Aus diesem Grund haben wir SQL Island [SD13] entwickelt. Mit diesem Spiel können die Schüler SQL lernen und Spaß dabei haben. Mittlerweile wird das Spiel zusätzlich auch in Datenbankvorlesungen als unterstützende Übung eingesetzt, sodass auch Studenten ihre SQL-Kenntnisse damit auffrischen und überprüfen können. Im Informatikunterricht an Schulen dient das Spiel bei Schülern ohne Programmiererfahrung vor allem dazu, zu erkennen, dass man bei einer Sprache, die der Computer verstehen soll, sehr präzise sein muss. Die Mischung aus Lehr-

[1] `http://technotag.uni-kl.de`

[2] `http://www.uni-kl.de/gleichstellung-und-familie/gleichstellung/massnahmen/aktivitaeten-zur-gewinnung-von-schuelerinnen/schuelerinnentag`

stoff, selbst SQL-Anfragen zu stellen und einer witzigen Story stößt sowohl bei Schülern als auch bei Studenten auf positive Resonanz.

2 Das SQL-basierte Text-Adventure

Zu Beginn dies Spiels ist die Spielfigur einziger Überlebender eines Flugzeugunglücks und landet auf einer Insel. Um von der Insel zu entkommen, muss diese zunächst erkundet werden. Ein Teil der im Spiel verwendeten SQL-Anfragen werden präsentiert, sodass der Spieler zunächst die Syntax kennenlernt und neue Befehle lernt: `SELECT * FROM dorf` ist eine Anfrage, um alle Dörfer der Insel zu sehen. Am unteren Bildschirmrand steht, welche Tabellen es gibt und welche Spalten diese haben. Mit diesen Informationen und der zuvor präsentierten Anfrage, ist der Benutzer in der Lage, die erste einfache Aufgabe zu lösen: „Welche Bewohner wohnen auf der Insel?". Die auf diese Frage korrekte Antwort `SELECT * FROM bewohner` wird im weiteren Spielverlauf immer weiter verfeinert, um nur Bewohner mit bestimmten Berufen zu finden und um deren Goldvorräte zu analysieren.

Im weiteren Spielverlauf muss die Spielfigur mittels UPDATE-Anfragen Gegenstände sammeln und diese verkaufen. Dadurch erhöhen sich die Goldvorräte des Spielers und es kann ein Schwert gekauft werden, mit dem am Ende des Spiels gegen Bösewichte gekäpft werden kann. Der Kampf erfolgt mittels DELETE-Kommandos. Zunächst müssen die Bösewichte jedoch mit Verbund- und GROUP-BY-Anfragen gefunden werden.

2.1 SELECT-Anfragen

Die Sprache SQL besteht aus der Data Definition Language (DDL), um Tabellen anzulegen und ein Schema zu definieren und der Data Manipulation Language (DML), um Zeilen in Tabellen einzufügen, zu ändern und zu löschen, und um Daten, die bestimmten Kriterien entsprechen, abzufragen. Für letztgenanntes verwendet man SELECT-Abfragen. Die DDL wird im Spiel nicht vermittelt, da die Tabellen zu Spielbeginn bereits existieren und da dies unserer Meinung nach über die Grundlagen hinaus geht.

Betrachten wir die folgende Aufgabe: „Wenn wir Gegenstände sammeln und verkaufen und zusätzlich noch als Bäcker arbeiten, können wir maximal so viel Gold bekommen, wie die Händler, Kaufmänner und Bäcker zusammen besitzen. Wie viel ist das?". Die im Programm eingespeicherte korrekte Lösung lautet: `SELECT SUM(gold) FROM bewohner WHERE beruf = 'Haendler' OR beruf = 'Kaufmann' OR beruf = 'Baecker'`. Die Korrektheit einer vom Benutzer eingegebenen Lösung kann natürlich nicht über eine exakte Übereinstimmung des Eingabe-Strings mit der Musterlösung erfolgen, da auch Variationen zu einem äquivalenten Ergebnis führen können, beispielsweise `SUM(B.gold) FROM bewohner B` oder `beruf IN ('Haendler', 'Kaufmann', 'Baecker')`. Daher erfolgt die Überprüfung der Korrektheit einer Anfrage Q gegenüber der Musterlösung L in zwei Schritten:

1. Überprüfung der syntaktischen Korrektheit von Q, indem sie auf dem Datenbanksystem ausgeführt wird. Bei Misserfolg wird die Fehlermeldung an den Benutzer

weitergegeben und eventuell um weitere Hinweise ergänzt.

2. Bei syntaktischer Korrektheit, Ausführung der Anfragen Q `EXCEPT` L sowie L `EXCEPT` Q. Kommt bei beiden die leere Menge raus, gibt es kein fehlendes und kein zu viel ausgegebenes Tupel. Werden die falschen Spalten ausgewählt, ist die Subtraktionsverträglichkeit verletzt und die Anfrage führt zu einem Fehler.

2.2 INSERT-, DELETE- und UPDATE-Anfragen

Anfragen vom Typ `INSERT`, `DELETE` und `UPDATE` werden über eine Verfikationsanfrage sowie der erwarteten Kardinalität dieser Anfrage auf ihre Richtigkeit überprüft. Die Aufgabe „Lasst uns die Kaffeetasse einsammeln!" hat die Verfikationsanfrage `SELECT * FROM gegenstand WHERE besitzer=20 AND gegenstand='Kaffeetasse'`. Die erwartete Kardinalität ist 1. Liefert nach Ausführung der Benutzer-Anfrage die Verifikation die korrekte Anzahl an Zeilen, gilt die Aufgabe als gelöst.

2.3 Verwendete Technologien

SQL Island ist eine Web-Anwendung, die mit PHP, JavaScript und HTML entwickelt wurde. Als Datenbanksystem kommt SQLite zum Einsatz, da dies Datei-basiert ist und somit für jeden Spieldurchlauf einfach eine neue Datenbankinstanz angelegt werden kann. Dies ermöglicht zum einen, dass sich parallel laufende Spiele nicht beeinflussen, da sie jeweils auf ihrer eigenen Instanz arbeiten. Zum anderen wird so verhindert, dass bösartige Anfragen oder SQL Injections das Spiel zerstören können. Ein Neustart des Spiels legt jedes mal das Schema samt Daten neu an. Aus Sicherheitsgründen werden zudem nur einfache SELECT-, INSERT-, UPDATE- und DELETE-Anfragen unterstützt, damit keine neuen Tabellen erstellt werden können, keine Systeminfos ausgelesen werden und damit eine rekursive INSERT-Anfrage, die im SELECT-Teil die Daten ermittelt, die sie gerade eingefügt hat, nicht möglich sind. Des Weiteren wird die Größe der Datenbankdatei überwacht und bei Erreichen einer Maximalgröße das Spiel beendet. Diese Größe beträgt nur wenige Kilobyte, jedoch das doppelte der Größe zu Spielbeginn, sodass bei einem normalen Spielverlauf dieser Fehler nicht auftritt.

3 Didaktische Betrachtung

Das Lernziel unseres Spiels ist der Kompetenzgewinn, um SQL-Anfragen zu verstehen und zu formulieren. Vergleicht man SQL Island mit dem klassischen Weg, SQL zu lernen, nämlich mit Literatur und Vorlesungen, sind die zwei großen Vorteile, dass beim spielbasierten Lernen durch Witz und einer Story die Lernenden motiviert werden, und dass diese Lernform interaktiv ist. Weitere Learning-by-Doing-Ansätze existieren sowohl für SQL als auch für andere Programmier- und Scriptsprachen. Einige Beispiele dafür sind *SQL-coach* der Fachhochschule Kaiserslautern [FH 13a], das eLearning-Datenbank-Portal *edb* der Fachhochschule Köln [FH 13b] sowie die SQL-Spielwiese *SQL Fiddle* [SQL13]. Die beiden erstgenannten Programme erlauben SELECT-Anfragen auf mit Testdaten gefüllten

Tabellen. Bei SQL Fiddle müssen die Tabellen selbst angelegt und gefüllt werden. Dies kann per CREATE- und INSERT-Befehlen oder mittels CVS-Import erfolgen. Das Lernen im Spiel SQL Island ist jedoch, im Gegensatz zu den anderen Lernprogrammen, spielbasiert. In [MS03] werden verschiedene Arten von Lernspielen vorgestellt: vom Quiz, über virtuelle Lernwelten, bis hin zu Spielen mit unbemerktem Lernen. SQL Island fällt in die Kategorie des computerbasierten Trainings mit Spielelementen. Besonderheiten dieser Art von Spielen sind, dass der Benutzer selbst das Spielgeschehen beeinflusst, der Spaß, die Motivation, das Spiel zu gewinnen sowie eine interessante und witzige Story.

4 Demonstation

Die Demonstration besteht aus einem Poster, welches Informationen, Hintergründe und Implementierungsdetails zu SQL Island enthält, sowie das Spiel selbst. Besucher der Demo können das Spiel am PC spielen und Aufgaben lösen. Gerne können auch falsche Anfragen an die Datenbank gestellt werden, um das Spielverhalten zu beobachten. Da das Spiel online frei verfügbar ist, kann es gerne in Lehrveranstaltungen und Übungen eingesetzt werden. Für Fragen und Diskussionen über Anwendungsbereiche, Erweiterungen und über spielbasiertes Lernen in der Informatik stehen wir gerne zur Verfügung.

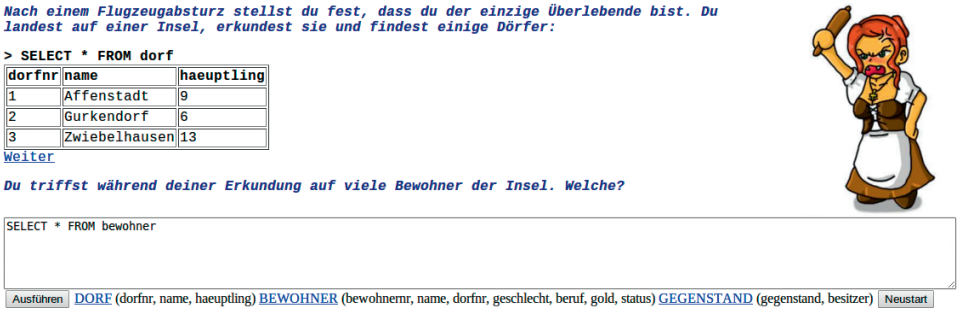


Abbildung 1: SQL Island - Screenshot

Literatur

[FH 13a] FH Kaiserslautern - Zweibrücken. SQLcoach, 2013. `http://sqlcoach.informatik.fh-kl.de/`.

[FH 13b] FH Köln. edb - Das eLearning DatenbankPortal, 2013. `http://edb.gm.fh-koeln.de`.

[MS03] Christoph Meier und Sabine Seufert. Game-based learning: Erfahrungen mit und Perspektiven für digitale Lernspiele in der beruflichen Bildung. *Grundlagen der Weiterbildung. Praxishilfen. Neuwied, Kriftel, Berlin*, 2003.

[SD13] Johannes Schildgen und Stefan Deßloch. „Gib mir so viel Gold, wie die Metzger im Nachbardorf zusammen besitzen und ich lasse den Piloten frei!“ – Spielbasiertes Lernen von SQL-Grundlagen. *Datenbank-Spektrum*, 13(3):243–249, 2013.

[SQL13] SQLFiddle.com. A tool for easy online testing and sharing of database problems and their solutions., 2013. `http://sqlfiddle.com`.

Visualizing the Behavior of an Elastic, Energy-Efficient Database Cluster

Sandy Ganza, Thomas Psota, Daniel Schall, Theo Härder
{s_ganza, t_psota, schall, haerder}@cs.uni-kl.de
Databases and Information Systems Group
University of Kaiserslautern, Germany

Abstract: Energy efficiency in databases is an emerging topic. Our research prototype *WattDB* dynamically adjusts the number of active servers in a cluster to the current workload to achieve energy proportionality. In this demo, we give insights in the partitioning process and *WattDB's* reaction to workload changes by live-presenting a monitoring GUI. The whole process and the resulting configuration are visualized to give immediate feedback, how the cluster would react.

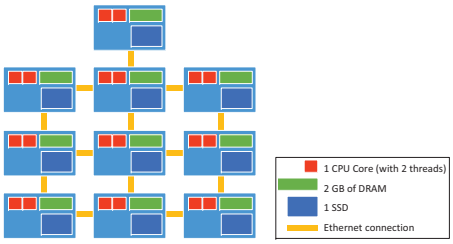
1 Introduction

Today's server hardware is not energy proportional; at low utilization, hardware—mainly main memory and storage drives—consumes a significant amount of power. Hence, about half of the maximum server power is already going to waste when idle. Scaling systems automatically down when idle, thus preventing high idle power consumption, is the main focus of today's servers to provide better energy efficiency. However, this is not enough to approximate energy proportionality.

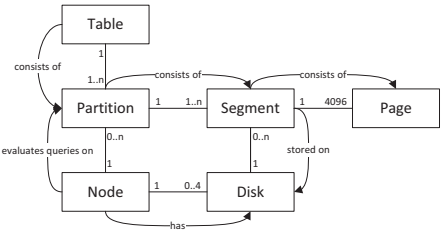
With cloud computing, elastic systems have emerged that adapt their size to the current workload. While stateless or lightweight systems can easily increase or reduce the number of active computing nodes in a cluster, a database faces much more challenges due to the need of high interactions among the nodes and fast reachability of all DB data.

Similar to cloud-based solutions, we hypothesize that a cluster of nodes may adjust the number of active (power-consuming) nodes to the current demand and, thus, approximate energy proportionality. Based on these observations, we developed WattDB, a research prototype of a distributed DBMS cluster, running on lightweight, Amdahl-balanced nodes using commodity hardware. The cluster is intended to dynamically shrink and expand its size, dependent on the workload. Reconfiguring a cluster to dynamically match the workload requires data to be moved from node to node to balance utilization. Yet, copying data is time-consuming and adds overhead to the already loaded cluster. But reducing both, time and overhead, is crucial for an elastic DBMS.

In this demo, we present our recent and ongoing work with WattDB. To give a better understanding of the internal process of balancing and decision making, we show some insights of our DBMS.



(a) The 10-node cluster



(b) Database schema

2 The WattDB Approach

Traditional clustered DBMSs do not dynamically adjust their size (in terms of the number of active nodes) to their workload. Hence, scale-out to additional nodes is typically supported, whereas the opposite functionality, shrinking the cluster and centralizing the processing—the so-called scale-in—, is not. Recently, with the emergence of clouds, a change of thinking occurred and dynamic solutions became a research topic. To test our hypothesis of an elastic, energy-proportional DBMS, that dynamically scales out and back in and that supports SQL query processing with ACID properties, we developed *WattDB*.

The smallest configuration of WattDB is a single server, hosting all database functions and acting as endpoint to DB clients. This server is called *master node*. DB objects (tables, partitions) and query evaluation can be offloaded to arbitrary nodes in the cluster to relieve overloaded nodes, but the master will always act as the coordinator and client endpoint.

To run queries on a cluster of nodes, distributed query plans are generated at the master node. Except data access operators which need local access to the database's records, all query operators can be placed on remote nodes. Running query operators on a single node does not involve network communication among query operators, because all records are transferred via main memory. In contrast, distributing operators implies shipping of records among nodes and, hence, introduces network latencies. Additionally, the bandwidth of the Gigabit Ethernet, which we are using for our experiments, is relatively small, compared to memory bandwidth.

The master node is coordinating the whole cluster. It is globally optimizing the query plans, whereas regular nodes can locally optimize their part of the plan. Furthermore, it takes nodes on- and offline and decides when and how DB tables are (re)partitioned.

Every node is monitoring its utilization: CPU, memory use, network I/O, and disk utilization (storage and IOPS). Additionally, performance-critical data is collected for each DB partition, i. e., CPU cycles, buffer page requests, and network I/O. With these figures, we can correlate the observed utilization of cluster components to (logical) DB entities.

The master checks the incoming performance data to predefined thresholds—with both upper and lower bounds. If an overloaded component is detected, it will decide where to distribute data and whether to power on additional nodes and resume their cluster participation. Similar, underutilized nodes trigger a scale-in protocol, i. e., the master will distribute the data (processing) to fewer nodes and shutdown the nodes currently not needed.

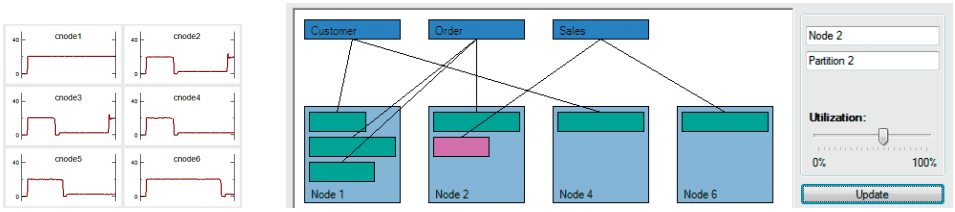
In WattDB, we provide different policies regarding the scale-out behavior. First, each node in the cluster stores data on local disks to minimize network communication. If storage space of a node is in short supply, DB partitions are split up on nodes with free space. In this way, it tries to keep the I/O rate for each storage disk in a certain range. Underutilized disks are eligible for additional data—either newly generated by INSERT operations or migrated from overloaded disks. Utilization among storage disks is first locally balanced on each node, before an allocation of data from/to other nodes is considered. Third, each node's CPU utilization should not exceed the upper bound of the specified threshold (80%) for more than short utilization spikes. As soon as this bound is violated on a node for longer than one minute, WattDB first tries to offload query processing to underutilized nodes.[1] If the overload situation cannot be resolved by redistributing the query load, the current data partitions and their node assignments are reconsidered. When a partition causing the overload is identified, it is split according to the partitioning scheme applied, where affected segments are moved to other nodes [SH15]. For underutilized nodes, an inverse approach is needed. A scale-in protocol is initiated, which quiesces the involved nodes from query processing and shifts their data partitions to nodes currently having sufficient processing capacity.

Similar rules exist for network and memory utilization, e. g., if the working sets of the transactions become too big for the DB buffer, repartitioning is triggered. The master makes decisions based on the current workload, the course of utilization in the recent past, and the expected future workloads [KHH12]. Additionally, workload shifts can be user-defined to inform the cluster of an expected change in utilization.

WattDB utilizes a rather simplistic DB schema with logically partitioned tables. A *table* is a purely logical construct in WattDB. Its metadata (column definitions, partitioning scheme) is maintained on the master node. Each table is composed of k *horizontal partitions*, each belonging to a specific node, responsible for query evaluation, data integrity (logging), and access synchronization (locking). Each partition contains 1 to m *segments* (of 32 MB), which are physical units of storage. Each segment is located on a local disk to reduce access latencies. Partitions are by default index-organized [Sea00] w. r. t. the primary key and can be supported by additional, secondary indexes. Fig. 1b clarifies these terms and their relationships.

To facilitate speedy rebalancing, we have implemented *physiological partitioning* [SH15] to be able to quickly repartition the cluster in case of workload imbalances. With physiological partitioning, each segment is a self-contained subpartition, including a defined primary-key range. The partition's metadata only stores a list of segments with their respective primary-key ranges. Therefore, segments can be detached from one partition and copied to another node, with minimal impact to query processing. Since segments are self-contained, no outside references need to be changed during rebalancing, except updating metadata in the partition and on the master. We have experimentally verified the advantages of physiological partitioning vs. other approaches and conclude that our system is well-suited for an elastic DB cluster. Yet, details about the repartitioning process inside the master node cannot be discussed here [SH15]. In this demo, we exhibit how physiological partitioning allows the master to quickly rebalance the cluster.

[1] This works well for operators like **SORT**, **GROUP**, and **AGGREGATE**.



(a) Power consumption

(b) Partitioning and utilization

Figure 2: Graphical user interface

3 Demo Description

We are going to present how partitioning in WattDB works and how workload changes trigger rebalancing of DB partitions. We put up a GUI, showing the live status of the cluster (via remote connection). Demo visitors will see the hardware state of the nodes, i. e., whether they are online or offline, and their corresponding power consumption. Additionally, a predefined DB schema with its tables and partitions is sketched. The assignment of partitions to nodes and the current utilization of nodes and partitions is also visible. Hence, the GUI will show both, the underlying hardware as well as the DB layout on top.

Users will be able to alter the workloads by increasing or reducing utilization of tables and partitions. The new workload profile is sent to WattDB, where the master node will then trigger rebalancing. The audience can see, how the cluster rebalances, moves data, and potentially splits and merges partitions to adjust to the modified workload. The master node will power up additional nodes, if necessary, which will affect overall power consumption. Likewise, superfluous nodes are shut down and the outcome is visible on the GUI.

Since rebalancing data is time-consuming and we do not expect visitors to wait several minutes, the master nodes does not send out actual rebalancing commands to the nodes. Instead, only the process on the master node is depicted, showing how WattDB's decision process works and what partitions are deemed candidates for rebalancing. Hence, the audience will be able to see the reaction of the cluster almost instantaneously.

References

[KHH12] Christopher Kramer, Volker Höfner, and Theo Härder. Lastprognose für energieeffiziente verteilte DBMS. *Proc. 42. GI-Jahrestagung 2012, LNI 208*, pages 397–411, 2012.

[Sea00] Jagannathan Srinivasan and et al. Oracle8I Index-Organized Table and Its Application to New Domains. In *Proc. VLDB Conf.*, pages 285–296, 2000.

[SH15] Daniel Schall and Theo Härder. Dynamic Physiological Partitioning on a Shared-nothing Database Cluster. In *ICDE Conf.*, 2015.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Lan-guage to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Er-fahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting. CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik –
Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^{2}CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-182 Ronald Maier (Hrsg.)
6th Conference on Professional Knowledge Management
From Knowledge to Action

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.)
Software Engineering 2011
Fachtagung des GI-Fachbereichs Softwaretechnik

P-184 Ralf Reussner, Alexander Pretschner, Stefan Jähnichen (Hrsg.)
Software Engineering 2011
Workshopband
(inkl. Doktorandensymposium)

P-185 Hagen Höpfner, Günther Specht, Thomas Ritz, Christian Bunse (Hrsg.)
MMS 2011: Mobile und ubiquitäre Informationssysteme Proceedings zur 6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2011)

P-186 Gerald Eichler, Axel Küpper, Volkmar Schau, Hacène Fouchal, Herwig Unger (Eds.)
11th International Conference on Innovative Internet Community Systems (I^2CS)

P-187 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
4. DFN-Forum Kommunikations-technologien, Beiträge der Fachtagung 20. Juni bis 21. Juni 2011 Bonn

P-188 Holger Rohland, Andrea Kienle, Steffen Friedrich (Hrsg.)
DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. 5.–8. September 2011, Dresden

P-189 Thomas, Marco (Hrsg.)
Informatik in Bildung und Beruf
INFOS 2011
14. GI-Fachtagung Informatik und Schule

P-190 Markus Nüttgens, Oliver Thomas, Barbara Weber (Eds.)
Enterprise Modelling and Information Systems Architectures (EMISA 2011)

P-191 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2011
International Conference of the Biometrics Special Interest Group

P-192 Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, Jörg Schneider (Hrsg.)
INFORMATIK 2011
Informatik schafft Communities

P-193 Wolfgang Lehner, Gunther Piller (Hrsg.)
IMDM 2011

P-194 M. Clasen, G. Fröhlich, H. Bernhardt, K. Hildebrand, B. Theuvsen (Hrsg.)
Informationstechnologie für eine nachhaltige Landbewirtschaftung
Fokus Forstwirtschaft

P-195 Neeraj Suri, Michael Waidner (Hrsg.)
Sicherheit 2012
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)

P-196 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2012
Proceedings of the 11th International Conference of the Biometrics Special Interest Group

P-197 Jörn von Lucke, Christian P. Geiger, Siegfried Kaiser, Erich Schweighofer, Maria A. Wimmer (Hrsg.)
Auf dem Weg zu einer offenen, smarten und vernetzten Verwaltungskultur
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2012

P-198 Stefan Jähnichen, Axel Küpper, Sahin Albayrak (Hrsg.)
Software Engineering 2012
Fachtagung des GI-Fachbereichs Softwaretechnik

P-199 Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)
Software Engineering 2012
Workshopband

P-200 Gero Mühl, Jan Richling, Andreas Herkersdorf (Hrsg.)
ARCS 2012 Workshops

P-201 Elmar J. Sinz Andy Schürr (Hrsg.)
Modellierung 2012

P-202 Andrea Back, Markus Bick, Martin Breunig, Key Pousttchi, Frédéric Thiesse (Hrsg.)
MMS 2012:Mobile und Ubiquitäre Informationssysteme

P-203 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
5. DFN-Forum Kommunikations-technologien
Beiträge der Fachtagung

P-204 Gerald Eichler, Leendert W. M. Wienhofen, Anders Kofod-Petersen, Herwig Unger (Eds.)
12th International Conference on Innovative Internet Community Systems (I2CS 2012)

P-205 Manuel J. Kripp, Melanie Volkamer, Rüdiger Grimm (Eds.)
5th International Conference on Electronic Voting 2012 (EVOTE2012)
Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-206 Stefanie Rinderle-Ma, Mathias Weske (Hrsg.)
EMISA 2012
Der Mensch im Zentrum der Modellierung

P-207 Jörg Desel, Jörg M. Haake, Christian Spannagel (Hrsg.)
DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.
24.–26. September 2012

P-208 Ursula Goltz, Marcus Magnor, Hans-Jürgen Appelrath, Herbert Matthies, Wolf-Tilo Balke, Lars Wolf (Hrsg.) INFORMATIK 2012

P-209 Hans Brandt-Pook, André Fleer, Thorsten Spitta, Malte Wattenberg (Hrsg.) Nachhaltiges Software Management

P-210 Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller, Silke Spitzer (Hrsg.) Automotive – Safety & Security 2012 Sicherheit und Zuverlässigkeit für automobile Informationstechnik

P-211 M. Clasen, K. C. Kersebaum, A. Meyer-Aurich, B. Theuvsen (Hrsg.) Massendatenmanagement in der Agrar- und Ernährungswirtschaft Erhebung - Verarbeitung - Nutzung Referate der 33. GIL-Jahrestagung 20. – 21. Februar 2013, Potsdam

P-212 Arslan Brömme, Christoph Busch (Eds.) BIOSIG 2013 Proceedings of the 12th International Conference of the Biometrics Special Interest Group 04.–06. September 2013 Darmstadt, Germany

P-213 Stefan Kowalewski, Bernhard Rumpe (Hrsg.) Software Engineering 2013 Fachtagung des GI-Fachbereichs Softwaretechnik

P-214 Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mit schang, Theo Härder, Veit Köppen (Hrsg.) Datenbanksysteme für Business, Technologie und Web (BTW) 2013 13. – 15. März 2013, Magdeburg

P-215 Stefan Wagner, Horst Lichter (Hrsg.) Software Engineering 2013 Workshopband (inkl. Doktorandensymposium) 26. Februar – 1. März 2013, Aachen

P-216 Gunter Saake, Andreas Henrich, Wolfgang Lehner, Thomas Neumann, Veit Köppen (Hrsg.) Datenbanksysteme für Business, Technologie und Web (BTW) 2013 – Workshopband 11. – 12. März 2013, Magdeburg

P-217 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.) 6. DFN-Forum Kommunikations- technologien Beiträge der Fachtagung 03.–04. Juni 2013, Erlangen

P-218 Andreas Breiter, Christoph Rensing (Hrsg.) DeLFI 2013: Die 11 e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. (GI) 8. – 11. September 2013, Bremen

P-219 Norbert Breier, Peer Stechert, Thomas Wilke (Hrsg.) Informatik erweitert Horizonte INFOS 2013 15. GI-Fachtagung Informatik und Schule 26. – 28. September 2013

P-220 Matthias Horbach (Hrsg.) INFORMATIK 2013 Informatik angepasst an Mensch, Organisation und Umwelt 16. – 20. September 2013, Koblenz

P-221 Maria A. Wimmer, Marijn Janssen, Ann Macintosh, Hans Jochen Scholl, Efthimios Tambouris (Eds.) Electronic Government and Electronic Participation Joint Proceedings of Ongoing Research of IFIP EGOV and IFIP ePart 2013 16. – 19. September 2013, Koblenz

P-222 Reinhard Jung, Manfred Reichert (Eds.) Enterprise Modelling and Information Systems Architectures (EMISA 2013) St. Gallen, Switzerland September 5. – 6. 2013

P-223 Detlef Hühnlein, Heiko Roßnagel (Hrsg.) Open Identity Summit 2013 10. – 11. September 2013 Kloster Banz, Germany

P-224 Eckhart Hanser, Martin Mikusz, Masud Fazal-Baqaie (Hrsg.) Vorgehensmodelle 2013 Vorgehensmodelle – Anspruch und Wirklichkeit 20. Tagung der Fachgruppe Vorgehensmodelle im Fachgebiet Wirtschaftsinformatik (WI-VM) der Gesellschaft für Informatik e.V. Lörrach, 2013

P-225 Hans-Georg Fill, Dimitris Karagiannis, Ulrich Reimer (Hrsg.) Modellierung 2014 19. – 21. März 2014, Wien

P-226 M. Clasen, M. Hamer, S. Lehnert, B. Petersen, B. Theuvsen (Hrsg.) IT-Standards in der Agrar- und Ernährungswirtschaft Fokus: Risiko- und Krisenmanagement Referate der 34. GIL-Jahrestagung 24. – 25. Februar 2014, Bonn

P-227 Wilhelm Hasselbring,
Nils Christian Ehmke (Hrsg.)
Software Engineering 2014
Fachtagung des GI-Fachbereichs Softwaretechnik
25. – 28. Februar 2014
Kiel, Deutschland

P-228 Stefan Katzenbeisser, Volkmar Lotz, Edgar Weippl (Hrsg.)
Sicherheit 2014
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
19. – 21. März 2014, Wien

P-230 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2014
Proceedings of the 13th International Conference of the Biometrics Special Interest Group
10. – 12. September 2014 in Darmstadt, Germany

P-231 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
7. DFN-Forum Kommunikationstechnologien
16. – 17. Juni 2014
Fulda

P-232 E. Plödereder, L. Grunske, E. Schneider, D. Ull (Hrsg.)
INFORMATIK 2014
Big Data – Komplexität meistern
22. – 26. September 2014
Stuttgart

P-233 Stephan Trahasch, Rolf Plötzner, Gerhard Schneider, Claudia Gayer, Daniel Sassiat, Nicole Wöhrle (Hrsg.)
DeLFI 2014 – Die 12. e-Learning Fachtagung Informatik
der Gesellschaft für Informatik e.V.
15. – 17. September 2014
Freiburg

P-234 Fernand Feltz, Bela Mutschler, Benoît Otjacques (Eds.)
Enterprise Modelling and Information Systems Architectures
(EMISA 2014)
Luxembourg, September 25-26, 2014

P-235 Robert Giegerich, Ralf Hofestädt, Tim W. Nattkemper (Eds.)
German Conference on Bioinformatics 2014
September 28 – October 1
Bielefeld, Germany

P-236 Martin Engstler, Eckhart Hanser, Martin Mikusz, Georg Herzwurm (Hrsg.)
Projektmanagement und Vorgehensmodelle 2014
Soziale Aspekte und Standardisierung
Gemeinsame Tagung der Fachgruppen Projektmanagement (WI-PM) und Vorgehensmodelle (WI-VM) im Fachgebiet Wirtschaftsinformatik der Gesellschaft für Informatik e.V., Stuttgart 2014

P-237 Detlef Hühnlein, Heiko Roßnagel (Hrsg.)
Open Identity Summit 2014
4.–6. November 2014
Stuttgart, Germany

P-238 Arno Ruckelshausen, Hans-Peter Schwarz, Brigitte Theuvsen (Hrsg.)
Informatik in der Land-, Forst- und Ernährungswirtschaft
Referate der 35. GIL-Jahrestagung
23. – 24. Februar 2015, Geisenheim

P-241 Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, Wolfram Wingerath (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW 2015)
04. – 06. März 2015, Hamburg

P-242 Norbert Ritter, Andreas Henrich, Wolfgang Lehner, Andreas Thor, Steffen Friedrich, Wolfram Wingerath (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW 2015) – Workshopband
02. – 03. März 2015, Hamburg

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de

Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-635-0

The BTW 2015 in Hamburg is the 16th conference of its kind reflecting the broad range of academic research and industrial development work within the German database community. Some of the various hot topics of this 2015 conference are: new hardware and memory technologies, information extraction, information integration, big data analytics, web data management, service-oriented computing and cloud computing. This volume contains contributions from the refereed scientific program, the refereed industrial program, and the refereed demo program. Furthermore, the dissertation award winners and keynotes are presented in this volume.