Multi-Schema-Version Data Management

Herrmann, Kai

DOI (link to publication from Publisher):
10.5278/vbn.phd.tech.00021

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

MULTI-SCHEMA-VERSION DATA MANAGEMENT

BY
KAI HERRMANN

DISSERTATION SUBMITTED 2017

AALBORG UNIVERSITY
DENMARK

Multi-Schema-Version Data Management

Dissertation

submitted August 31, 2017

by **Dipl.-Inf. Kai Herrmann**
born July 25, 1988 in Dresden

at Technische Universität Dresden
and Aalborg Universitet

Supervisors:
Prof. Dr.-Ing. Wolfgang Lehner
Prof. Torben Bach Pedersen

Dissertation submitted:	September 2017
TUD Ph.D. Supervisor:	Prof. Dr.-Ing. Wolfgang Lehner Technische Universität Dresden, Dresden, Germany
AAU Ph.D. Supervisor:	Prof. Torben Bach Pedersen Aalborg University, Aalborg, Denmark
TUD Ph.D. Committee:	Prof. Dr. Franz Baader Technische Universität Dresden, Dresden, Germany
	Prof. Dr.-Ing. Wolfgang Lehner Technische Universität Dresden, Dresden, Germany
	Prof. Dr.-Ing. Stefan Deßloch Technische Universität Kaiserslautern, Kaiserslautern, Germany
	Prof. Dr. Thorsten Strufe Technische Universität Dresden, Dresden, Germany
	Prof. Torben Bach Pedersen Aalborg University, Aalborg, Denmark
AAU Ph.D. Committee:	Associate Professor Kristian Torp (chairman) Aalborg University, Denmark
	Prof. Dr.-Ing. Stefan Deßloch University of Kaiserslautern, Germany
	Associate Professor Panos Vassiliadis University of Ioannina, Greece
PhD Series:	Technical Faculty of IT and Design, Aalborg University
Institut:	Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-074-6

Published by:
Aalborg University Press
Skjernvej 4A, 2nd floor
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2017

The thesis is jointly submitted to the Faculty of Computer Science at Technische Universität Dresden (TUD) and the Technical Faculty of IT and Design at Aalborg University (AAU), in partial fulfillment of the requirements within the scope of the IT4BIDC program for the joint Ph.D. degree in computer science (TUD: Dr.-Ing., AAU: Ph.D. in computer science). The thesis is not submitted to any other organization at the same time.

THESIS DETAILS

Thesis Title:	Multi-Schema-Version Data Management
Ph.D. Student:	Kai Herrmann
Supervisors:	Prof. Dr.-Ing. Wolfgang Lehner, Technische Universität Dresden
	Prof. Torben Bach Pedersen, Aalborg University

The main body of this thesis consists of the following peer-reviewed and published papers.

1. Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *International Conference on Management Of Data (SIGMOD)*, pages 1101–1116. ACM, 2017
   Input for Chapters 2 and 6

2. Kai Herrmann, Hannes Voigt, Jonas Rausch, Andreas Behrend, and Wolfgang Lehner. Robust and simple database evolution. *Information Systems Frontiers*, (Special issue):1–17, 2017
   Input for Chapters 4 and 5

3. Kai Herrmann, Hannes Voigt, Thorsten Seyschab, and Wolfgang Lehner. InVerDa – The Liquid Database. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 619–622. Gesellschaft für Informatik (GI), 2017
   Relates to Chapter 2

4. Kai Herrmann, Hannes Voigt, Thorsten Seyschab, and Wolfgang Lehner. InVerDa – co-existing schema versions made foolproof. In *International Conference on Data Engineering (ICDE)*, pages 1362–1365. IEEE, 2016
   Relates to Chapter 2

5. Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. CoDEL – A Relationally Complete Language for Database Evolution. In *European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 63–76. Springer, 2015
   Relates to Chapter 4

This thesis has been submitted for assessment in partial fulfillment of the joint Ph.D. degree. The thesis is based on the published scientific papers which are listed above. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Technical Doctoral School of IT and Design at Aalborg University and the Faculty of Computer Science at Technische Universität Dresden.

ABSTRACT

Modern agile software development methods allow to continuously evolve software systems by easily adding new features, fixing bugs, and adapting the software to changing requirements and conditions while it is continuously used by the users. A major obstacle in the agile evolution is the underlying database that persists the software system's data from day one on. Hence, evolving the database schema requires to evolve the existing data accordingly—at this point, the currently established solutions are very expensive and error-prone and far from agile.

In this thesis, we present InVerDa, a multi-schema-version database system to facilitate agile database development. Multi-schema-version database systems provide multiple schema versions within the same database, where each schema version itself behaves like a regular single-schema database. Creating new schema versions is very simple to provide the desired agility for database development. All created schema versions can co-exist and write operations are immediately propagated between schema versions with a best-effort strategy. Developers do not have to implement the propagation logic of data accesses between schema versions by hand, but InVerDa automatically generates it.

To facilitate multi-schema-version database systems, we equip developers with a relational complete and bidirectional database evolution language (BiDEL) that allows to easily evolve existing schema versions to new ones. BiDEL allows to express the evolution of both the schema and the data both forwards and backwards in intuitive and consistent operations; the BiDEL evolution scripts are orders of magnitude shorter than implementing the same behavior with standard SQL and are even less likely to be erroneous, since they describe a developer's intention of the evolution exclusively on the level of tables without further technical details. Having the developers' intentions explicitly given in the BiDEL scripts further allows to create a new schema version by merging already existing ones.

Having multiple co-existing schema versions in one database raises the need for a sophisticated physical materialization. Multi-schema-version database systems provide full data independence, hence the database administrator can choose a feasible materialization, whereby the multi-schema-version database system internally ensures that no data is lost. The search space of possible materializations can grow exponentially with the number of schema versions. Therefore, we present an adviser that releases the database administrator from diving into the complex performance characteristics of multi-schema-version database systems and merely proposes an optimized materialization for a given workload within seconds. Optimized materializations have shown to improve the performance for a given workload by orders of magnitude.

We formally guarantee data independence for multi-schema-version database systems. To this end, we show that every single schema version behaves like a regular single-schema database independent of the chosen physical materialization. This important guarantee allows to easily evolve and access the database in agile software development—all the important features of relational databases, such as transaction guarantees, are preserved. To the best of our knowledge, we are the first to realize such a multi-schema-version database system that allows agile evolution of production databases with full support of co-existing schema versions and formally guaranteed data independence.

DANSK RESUMÉ (SUMMARY IN DANISH)

Moderne agile software udviklings metoder gør det muligt at kontinuerligt videreudvikle software systemer ved at tilføje ny funktionalitet, fjerne fejl og tilpasse systemet til at opfylde nye krav samtidig med at det bliver brugt af dets brugere. En stor udfordring i den agile software udviklinger er den underliggende database der indeholder systemets data, fordi at foretage ændringer af database skemaet kræver også at alt data bliver ændret. At foretage sådanne ændringer i nuværende etablerede løsninger er meget ressourcekrævende og risikoen for fejl er høj og hele processen er langt fra agil.

I denne afhandling, præsentere vi InVerDa, en multi-skema-version database der muliggøre agil videreudvikling af databasen. Multi-skema-version databasesystemer har flere versioner af database skemaet, hvor hvert skema version fungere som en standard enkelt-skema databasesystem. Det er meget simpelt at lave en ny version af et skema og det er passer godt med den agil udviklings filosofi. Flere skema version kan eksistere samtidigt og skrive operationer bliver propageret mellem alle skemaer. Software udviklerne behøver ikke at implementere datatilgang logikken mellem skema versioner, det gør InVerDa automatisk. For at facilitere multi-skema-version systemer, har vi skabt et relationelt komplet og tovejs database udviklings sprog (BiDEL), det gør det muligt og nemt at lave nye og videreudvikle eksisterende skemaer. BiDEL gør det muligt at skifte mellem sema versioner både fremad og bagud i tid. Endvidere, er BiDEL sproget flere størrelsesordner kortere end implementationen af samme opførelse i standard SQL. Der er også langt mindre risiko for at lave fejl ved implementeringen fordi sproget beskriver software udviklerens intentioner, baseret på database tabeller, uden at gå i unødige tekniske detaljer. At have software udviklernes intentioner beskrevet med BiDEL gør det muligt at kombinere eksisterende skemaer for at skema helt nye skema versioner.

Det er at have flere database skema versioner i en database kræver sofistikeret fysisk materialisering af data. Multi-skema-version databasesystemer tager konceptet "data uafhængighed" til et højere niveau. Databaseadministratoren kan frit ændre den fysiske materialisering og systemet vil sikre at intet data er gået tabt. Søge rummet af mulige materialiseringer er eksponentielt voksende med antallet af skemaer. Derfor, har vi udviklet en rådgiver der frigør databaseadministratoren for at beskæftige sig med komplekse ydeevne karakteristikker ved at foreslå den optimale materialisering af skemaer for en givet arbejdsbelastning på få sekunder. Denne optimale materialisering forbedre ydeevnen for en givet arbejdsbelastning med flere størrelsesordner.

Vi kan formelt garantere at multi-skema-version databasesystemet er data uafhængigt, altså, vi viser at hver eneste schema version opfører sig som en standard enkelt-skema database, uafhængig af den valgte fysiske materialisering. Denne garanti gør det nemt at videreudvikle og tilgå databasen som en del af agil softwareudvikling, fordi all vigtige funktioner fra relationelle databaser, såsom transaktionelle garantier, er bevaret. Vi er de første til at realisere sådan et multi-skema-version databasesystemer der muliggøre en løbende udvikling af produktions databaser med fuld understøtning af skema versionering og formelt garantere data uafhængighed.

Thanks to Kim Ahlstrøm Meyn Mathiassen (AAU) for the translation.

ACKNOWLEDGEMENTS

I want to thank all my companions and supporters throughout my Ph.D. time for their direct or indirect contributions to this thesis. First, I want to thank Wolfgang Lehner, who gave me the opportunity to do research in my field of interest with perfect conditions. An essential part of these "perfect conditions" are all my colleagues that make up the productive and enjoyable work atmosphere with really cool scientific and less-scientific projects and discussions. Particularly, I thank Ahmad, Alex, Annett, Benjamin, Bernd, Bijay, Claudio, Dirk, Elvis, Frank, Hannes, Ines, Ismail, Johannes, Julian, Juliana, Kasun, Katrin, Lars, Lisa, Lucas, Maik, Martin, Mikhail, Muhammad, Patrick, Rana, Rihan, Robert, Till, Thomas, Tim, Tobias, Tomas, Uli, and Ulrike. I especially want to thank Hannes Voigt for the patient supervision, for the intense discussions, and for all the text editing to slowly but steadily improve my writing abilities. It is his merit that I actually enjoyed writing up this thesis—at least most of the time. I also thank Andreas Behrend, who was very supportive and contributed his expertise for building the solid formal ground of my work, as well as the students Jonas Rausch and Thorsten Seyschab for their excellent implementation and research work. Further, I thank all the Ph.D. students and investigators of the RoSI Research Training Group. I am happy and grateful to be a part of this group: Adrian, Alexander, Christiane, Christian, Christopher, Hendrik, İsmail, Jan, Johannes, Johannes, Philipp, Lars, Ly, Mandy, Markus, Mariam, Martin, Max, Romina, Steffen, Stephan, and Thomas. Especially the diversity of different chairs and topics creates a thrilling research environment from which I benefited a lot. Also, the evening events and the Ultimate Frisbee matches at our workshops are unforgettable.

Besides the database chair of Wolfgang Lehner and the RoSI group, I also had the opportunity to visit the group of my co-supervisor Torben Bach Pedersen at Aalborg University. I want to thank him for supervising and supporting me in Aalborg—and for introducing me to the Danish cuisine. The enjoyable stays in Aalborg with the great international team and the productive discussions definitively made up for the long travels and the very Danish weather. I especially want to thank Aamir, Alvis, Bezaye, Bhuvan, Christian, Dalia, Davide, Faisal, Gabriela, Ilkcan, Jilin, Johannes, Jovan, Katja, Kim, Laurynas, Lawan, Luis, Nuref, Robert, Rudra, and Søren. Special thanks go to Bijay, Nuref, and Robert for letting me stay at their places, to Kim for translating my abstract into Danish and for introducing me to the Danish life and beer culture, and to the whole team for exciting foosball and badminton matches as well as for the cool evenings in the city of Aalborg.

Following the Danish "Hygge" (work-life-balance), there had to be a lot of life to strike the balance with all the work for this thesis. The "Mobiles Expertenkommando" and all my friends from school and studies throw all their weight on the life-side—thank you guys. Among others, brewing was a great distraction from the thesis since we had both very annoying—e.g. 3000 pointlessly drilled holes—but also very joyful moments. In the recent years, climbing has become one of my favorites to unwind. I am very thankful to my climbing friends from "Mimimi Bouldern" for all the joint adventures in Sächsische Schweiz and on Roque Nublo, Sa Gubia, and Alpspitze as well as all the other cool spots. I owe special thanks to all friends that participated in the "Rächtschraipungstrinken". This is a young but promising tradition that goes back to a Dresden physicist from Cologne, whose greatest idea was to invite all his friends over to spell-check his pre-final thesis—for every remarkable mistake he stood a round. I was happy to continue this tradition and thanks to David, Hanna, Kilian, Linda, Mo, Stefan, Stephan, Thomas, and U, my finally submitted thesis contains 105 mistakes less.

Throughout the last three decades, I could always count on my family, especially on my parents and my brother Tim. I am very thankful for all the support and all the shared activities—particularly, all the celebrations and the Whitsun- and Easter holidays with the extended family.

I would have thanked her first and foremost, but then I probably would have forgotten the rest of my companions and supporters; so last but not least, special thanks to my beloved Linda! She is a great supporter of my work but also my greatest critic ("Hm, probably relevant but boring"). I am especially thankful for all the great moments we shared in the recent years and I am looking forward to all the upcoming joys and challenges that we will share.

CONTENTS

INTRODUCTION

1.1 MOTIVATION

Software systems are omnipresent in our daily life. Examples range from small sensors and whiteware in the household, over smartphones, smartwatches, smart home devices, or self-driving cars, up to big enterprise BI solutions. Since software systems are part of our daily life, they are directly exposed to the constantly changing conditions, requirements, and expectations towards them, which change just as often as people change their mind or their contexts or their rules. In other words: the only constant is the change [82].

Software needs to be aware of its current context. For instance, a self-driving car transporting people behaves differently than a car transporting construction waste. Smart homes can be in comfort-, party-, or emergency-mode, and a smart phone presents content differently when used for outdoor activities compared to a normal business day. Above this physical context, software systems are also exposed to different logical contexts, which comprise different cultures, countries, legislations, economic systems, etc. For instance, a self-driving car from France needs to change from the right to the left lane after passing the Channel Tunnel. In sum, software systems need to be aware of their situation and need specialized models and behavior. Software systems are **evolved to multiple variants** to serve all the different logical and physical contexts.

Software is not only omnipresent in logical and physical contexts but also over time; productive software systems are usually bug-fixed or extended but not replaced as a whole. Following the mantra "Evolution instead of Revolution" [18, 7], software systems often comprise e.g. decades-old legacy systems with brand new web front ends or innovative analytics pipelines. For instance, self-driving cars get upgrades to learn driving in cities instead of highways only, apply new speed limits, or improve the board-entertainment system. In sum, software systems are **evolved over time** in order to meet changing environments, requirements, or legislation and to incorporate new features, fix bugs, or improve non-functional features like performance, robustness, security, or safety.

The described software systems comprise multiple **versions** both from the evolution to variants and their evolution over time. In such software systems, it is very hard to orchestrate the evolution to new versions of all subsystems, since they are typically run by different stakeholders with different development cycle and different upgrade policies. For instance, the vendor of the image processing and traffic detection unit for self-driving cars changes the detection of vehicles in general to a more fine-grained detection of cars, trucks, motorbikes, and bicycles each having special properties. However, the vendor still needs to support the plain recognition of vehicles in order to be compatible to the original simple drive-control. Moreover, not only components of a software system rely on other versions of subsystems but also human users often decide to defer updates to stay with a familiar version a little longer. According to the technology adoption life cycle [16], the majority of users waits for the early innovators to test a new technology, before adopting it themselves and some laggards refuse any change as long as possible. In sum, an important requirement for software systems is that different **versions co-exist at the same time**.

The majority of software systems is backed by a database to persist all the application's states and data. The database is the single point of truth for all applications within the software system and gives persistence and transaction guarantees. In this position, the database is exposed to the same evolution as the software system as a whole. Developers evolve the database schema to different variants to match different physical and logical contexts and developers evolve the database schema over time to react to changing requirements, to fix bugs, or to add features. However, the database persists the data in a software system over time, so changes at the database schema level also require to evolve the currently existing data. In practice, this is a major bottleneck in IT projects; according to a study from 2011, 31% of the project's budget is spent for data migration tasks [66].

Let us summarize the requirements posed on software systems and databases so far: There are different variants of subsystems that adapt to the logical/physical context, all these variants evolve independently over time, and different versions have to co-exist at the same time. To achieve this, the software technology community established agile development methods to handle the continuous evolution and e.g. build management systems or software product lines to handle variants. Another promising approach is role-based software development, which is a new programming model that praises adaptation of variants to the current context and their continuous evolution [71, 78, 83, 79, 106].

Agile development methods make big upfront planning obsolete, which significantly speeds up the development process. While the fast development and management of multiple application versions is state-of-the-art, the same is hard at the database end, though. Typically, a relational database is used to persist the data within a software system and to guarantee the ACID properties. Therefore, developers have to sit down and analyze the information to be stored and carefully design a feasible database schema, which covers all the important data and ensures consistency. This fully contradicts the agile development of applications: Data of a running application is persisted according to the designed schema and changing this schema subsequently endangers the consistency of the already persisted data. Data is sticky and evolutions with schema changes also require the corresponding consistency-preserving evolution of the existing data. While the data is stored in exactly one schema version, the agile and continuous evolution of software systems requires co-existing versions, as discussed before. So, all schema versions—both the original one and the evolved versions—need to be fully accessible and represent the same conceptual data set, but in different schema versions. Unfortunately, current Database Management Systems (DBMSes) do not support the management of different schema versions properly; there is neither support to easily create, evolve, merge, or drop schema versions, nor to migrate the database physically to a specific schema version, nor to let them co-exist.

For practitioners, valuable tools, such as Liquibase[1], Rails Migrations[2], and DBmaestro Teamwork[3], help to manage schema versions outside the DBMS and generate SQL scripts for migrating to a new schema version. They mitigate data migration costs, but focus on the plain forward schema evolution; support for e.g. merging different schema versions or for co-existing schema versions is very limited.

To manage different schema versions and let them co-exist in the same database, developers are essentially forced to migrate a database completely in one haul to a new schema version. Keeping older schema versions or variants alive to continue serving all database clients independently from their adoption time is very costly and error-prone. Before and after a one-haul migration, manually written and maintained **delta code** is required. The term delta code refers to any implementation of propagation logic necessary to run an application with a schema version different to the version used by the database. Delta code may consist of view and trigger definitions in the database, propagation code in the database access layer of the application, or ETL jobs for update propagation for a database replicated in different schema versions. Either way, schema version management, physical migration, and co-existing schema versions require considerable development resources [66]. In short, handling different schema versions is very costly and error-prone and forces developers into less agility, longer release cycles, riskier big-bang migration, etc.

NoSQL systems attempt to provide a more flexible data model and thereby reduce the need for evolving the database in the first place [84]. Common solutions are role-based DBMSes like RSQL [70, 71], document stores like MongoDB [32] and CouchDB [9], key-value-stores like Amazon's Dynamo [44], wide-column stores like Google's BigTable [29] and Cassandra [81], etc. They all structure their data and this structure will evolve eventually [82, 102]—evolving a NoSQL store is typically even more cumbersome as the schema information is often spread out in the whole code.

[1] http://www.liquibase.org/
[2] http://guides.rubyonrails.org/migrations.html
[3] http://www.dbmaestro.com/

The consistency, integrity, and durability of a company's business data is just as mission critical as a high performance for providing this data. Relational databases provide all these features in a simple and robust manner—at least for common single-schema databases. This does not include the support of multiple schema versions within one database: Applying manually written delta code, to e.g. realize co-existing schema versions, breaks the robustness, endangers the data's correctness, and negatively impacts the performance. The current manual solutions are expensive and error-prone and call for a more sophisticated support of database evolution within the DBMS. Researchers are well aware of this research potential and published a large body of literature [98, 99, 97, 87] surveying the need and promising solution approaches for database evolution support.

We envision **multi-schema-version database systems**. Such DBMSes inherently support the evolution of databases, particularly the creation/evolution, management, and deployment of different schema versions, to keep the pace with flexible and agile software development. New schema versions are evolved from existing ones using a simple and intuitive Database Evolution Language (DEL). A DEL provides Schema Modification Operations (SMOs) that evolve both the schema and the data in intuitive and consistent steps, like e.g. adding a column or partitioning a table. To manage the set of different schema versions, we envision an easy-to-use Schema Versions Management Language (VML) to create, drop, and merge schema versions, to specify the degree of synchronization between schema versions, and to migrate a database physically. Particularly, multi-schema-version database systems allow a single database to have multiple co-existing schema versions that all can be accessed at the same time. New schema versions become immediately available. Applications can read and write data through any schema version concurrently; writes in one version are reflected in all other versions but each schema version itself behaves like a regular single-schema database. Old schema versions can be easily dropped without losing any data that is visible in other schema versions. Furthermore, the database administrator (DBA) can easily configure the primary materialization of the data; for convenience multi-schema-version database systems should also equip the DBA with an adviser that proposes an optimized materialization for the current workload. Changing the physical materialization of the co-existing schema versions does not affect the availability of any schema version but may significantly speed up query processing.

The envisioned DBMS-integrated database evolution support completely eliminates the need to write a single line of delta code and thereby makes database evolution and migration just as agile as application development but also as robust as traditional database development. Within this thesis, we focus on relational multi-schema-version database systems, since relational databases are very well-established and the de facto standard for data management with a wide range of powerful features and tool support. Nevertheless, the need for multi-schema-version database systems also emerges for all other kinds of structured data management—we leave the respective adaption of our concepts for future work.

1.2 CONTRIBUTIONS

In this thesis, we present the concepts of InVerDa (Integrated Versioning of Databases), a system that realizes the envisioned multi-schema-version database system. InVerDa lets developers create multiple schema versions within the same database. New schema versions are created from scratch, derived from existing ones using a comprehensive and formally sound DEL, or created by merging two existing schema versions. Further, InVerDa allows to configure the degree of data propagation between schema versions as well as to migrate and optimize the physical database schema. The underlying DEL, which we call BiDEL (Bidirectional DEL), is relationally complete and bidirectional, which facilitates the creation of evolved schema versions and the propagation of data accesses between co-existing schema versions within the same database. Alternatively, developers can create new schema

versions by merging existing ones. For this purpose, we present MIX (semi-automatic consolidation algorithm), which uses the rich semantics of BIDEL to semi-automatically merge the intentions of evolutions to different schema versions. The key for supporting multiple schema versions in a multi-schema-version database system is to decouple the physical table schema from the availability of the logical schema versions. We create a solid formal basis, showing that the physical data can be managed independently of the logical schema versions: no matter how often the developers or DBAs evolve and change the database, each schema version acts like a full-fledged single-schema database and no data is ever lost, so all the important features and characteristics of relational databases are maintained. Our detailed contributions are:

Architectural blueprint for system integration To the best of our knowledge, we are the first to present end-to-end support for co-existing schema versions in a multi-schema-version database system. INVERDA provides developers and DBAs with an intuitive interface to write and execute evolution/migration scripts and automatically generate all required delta code to manage and persist multiple schema versions at the same time. We show how to integrate the INVERDA concept into an existing DBMS by reusing existing database functionality as much as possible. (Chapter 2)

Relational complete database evolution language We provide a formal definition of the semantics of BIDEL's SMOs as well as an SQL-like syntax. We formally evaluate the relational completeness of BIDEL by showing that all operations of the relational algebra can be expressed with BIDEL. (Chapter 4)

Bidirectional database evolution language We introduce a bidirectional extension for BIDEL, which enriches the SMOs' semantics to translate data accesses between schema versions both forwards and backwards. We show that BIDEL requires orders of magnitude less code than evolutions and migrations manually written in SQL. (Chapter 4)

Semi-automatic variant co-evolution We realize semi-automated variant co-evolution based on BIDEL, which allows to create a new schema version by merging the intentions of two existing schema versions. (Chapter 5)

Co-existing schema versions INVERDA generates delta code from BIDEL-specified schema evolutions to allow reading and writing on all co-existing schema versions, each providing an individual view on the same shared data set. Each schema version is a full-fledged database schema, which transparently hides the versioning from the user. In the evaluation, we show that delta code generation is really fast (<1 s) and the generated delta code shows query performance comparable to handwritten delta code. (Chapter 6)

Data independence for co-existing schema versions All co-existing schema versions are fully accessible and independent of the physical materialization, which can be changed by the DBA and may also include redundancy. With a single-line migration command, the DBA can instruct INVERDA to run the physical data movement as well as the adaptation of all involved delta code. In any case, we formally evaluate and guarantee that each schema version behaves like a regular single-schema database. In the evaluation, we show that this data independence can be used to achieve performance improvements by orders of magnitude. (Chapter 6)

Adviser for optimization of physical table schema INVERDA allows to change the materialization to any potentially redundant subset of the evolution history. With growing evolution histories and diverse workload mixes it becomes increasingly complex for the DBA to determine the optimal physical materialization. To this end, INVERDA automates this task: the DBA merely triggers INVERDA to analyze the current workload and propose an optimized materialization, which can then be confirmed by the DBA to actually run the physical migration. (Chapter 7)

In the remainder of this thesis, we first discuss related work in Chapter 3. Afterwards, we present two ways to create new schema versions within the database: first by evolving an existing schema version with BiDEL in Chapter 4, and second by merging two existing schema versions in Chapter 5. Having multiple—potentially co-existing—schema versions within one database raises research questions on how to physically store the data and guarantee correctness for each single schema version. We answer those questions in Chapter 6 by exploiting BiDEL's bidirectionality for full data independence: the DBA can freely move the data along the schema version history and all schema versions stay fully alive. In chapter 7, we analyze the DBA's search space of physical materialization in detail and present a workload-dependent adviser. We apply InVerDa in many specific database evolution scenarios in Chapter 8 and discuss to which extent InVerDa is already a feasible solution and identify open promising research questions, before concluding the thesis in Chapter 9.

MULTI-SCHEMA-VERSION DATABASE SYSTEMS

In this chapter, we specify the goal of multi-schema-version data management based on a detailed requirement analysis and we derive a suitable architecture for InVerDa, which is our multi-schema-version database system. Therefore, we analyze typical evolutions in software systems in Section 2.1 and derive general requirements for agile database development. This serves as a basis for a detailed description of the envisioned multi-schema-version database systems in Section 2.2. Afterwards, we introduce the envisioned functionality of InVerDa with the help of an exemplary user scenario in Section 2.3. The envisioned database evolution support requires significant extensions to the DBMS architecture as, e.g., there is currently no mean for the management of multiple schema versions. To this end, we use the collected requirements and the envisioned functionality to extend a DBMS architecture for InVerDa and thereby facilitate multi-schema-version data management in Section 2.4.

2.1 EVOLVING SOFTWARE SYSTEMS

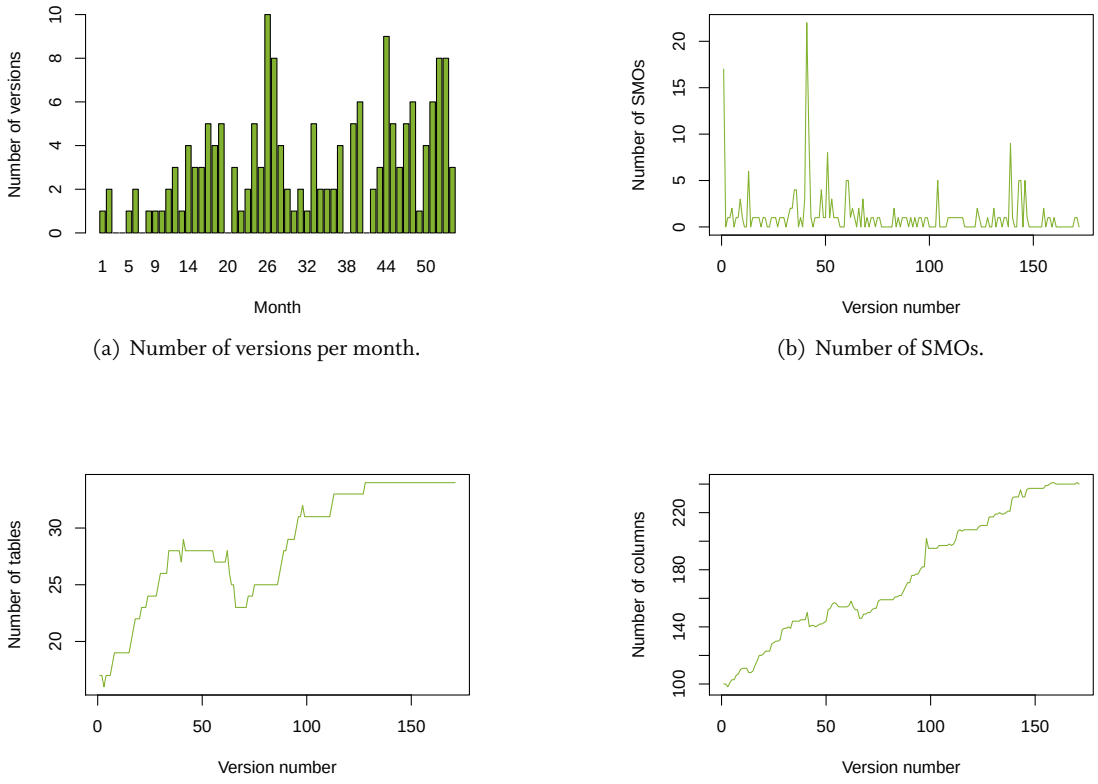
Software systems evolve continuously. This is a widely accepted fact these days but has already been formulated as laws back in the seventies by Meir M. Lehman [82]. These laws are based on the observation that software systems that solve real-world problems or are used for real-world applications need to be evolved continuously in order to maintain a high quality. Lehman's laws state that

- software has to evolve continuously, otherwise its quality appears to decrease,
- software becomes increasingly complex as it evolves, and
- the change rate is nearly constant over the life time of a software system.

These laws have been extensively reality-checked and revised by the research community [48, 122]. In a recent study from 2013, Liguo Yu [122] analyzed open source projects like Apache Tomcat[1] and Apache Ant[2] and found basically the same pattern as predicted by Lehman back in 1974. The software's quality (indicated by number of reported bugs) decreases when no effort is spent and can be kept on a constant level by continuously evolving the software.

Changes of the application layer usually require changes at the database layer and vice versa. In 2013, Qui analyzed the publicly available evolution history of ten open source projects and confirmed that 70% of all database schema changes are reflected in application code as well. The remaining changes comprise documentation, indices, etc. Specifically, each atomic evolution step at the database is connected to 10–100 changed lines of application code, where each database revision comprises two to five of these atomic changes on average [95]. The database of a software system follows Lehman's laws as well—with its own special characteristics. This has been analyzed by Skoulis in 2014 based on eight open source projects [104]. In contrast to the continuous evolution of software in general, databases face evolutions in bursts throughout their whole life time. The authors attribute this to the fact that changing the database schema may break running code and therefore is only applied if necessary and then in one haul. This greatly underpins the necessity for our envisioned database evolution support. Further, the growth in complexity is often not observable at the database layer—instead the size of database schemas increases a lot in early stages but in the analyzed projects it tends to stabilize after roughly five years. According to the authors, the database is still changed frequently in the latter phase, but developers are more careful and prefer to perform slighter changes as the maintenance of surrounding code gets increasingly complex and error-prone—again an observation calling for agile database evolution with multi-schema-version database systems.

[1] `http://tomcat.apache.org/`
[2] `http://ant.apache.org/`



(a) Number of versions per month.

(b) Number of SMOs.

(c) Number of tables.

(d) Number of columns.

Figure 2.1: Statistics about database schema evolution and database schema complexity.

As a prerequisite for identifying the correct and most important requirements for such multi-schema-version database systems, we dive into a deeper analysis of database evolution and figure out typical evolution pattern within the database. To this end, we analyze 171 schema versions of the evolution of Wikimedia[3], the backend of e.g. Wikipedia[4]. Thanks to Carlo A. Curino et al. [40], this evolution is easily available in the web and already analyzed to a large extent—however, he had a different focus and different definitions of database evolution steps. Hence, we conduct the analysis again but back it up with Curino's findings as well as with another study of eight open source projects performed by Vassiliadis et al. in 2015 [113].

Figure 2.1(a) shows the number of new schema versions per month between December 2003 and September 2007 [40]. As can be seen, the database schema continuously evolves; there are multiple new schema versions nearly every month throughout the whole life time. Further, Figure 2.1(b) shows the number of atomic schema modification operations (SMOs) per new schema version, so the schema is frequently changed. An SMO may e.g. add or delete a table/columns or partition or join tables. In fact, 55% of all evolutions contain SMOs while the remaining evolutions change documentation or indices; hence, evolving the database schema and the stored data is done frequently during software development. There are 209 SMOs in total (65% additive, 14% subtractive, 21% changing). As can be seen in Figure 2.1, the complexity of the database schema increases over time. In the Wikimedia example, the number of tables stabilizes temporarily after 120 schema versions (Figure 2.1(c)) while the number of columns continuously grows (Figure 2.1(d)). Thus, at a certain point all relevant concepts have been represented and merely these concepts are refined further.

[3] `https://www.wikimedia.de/`

[4] `https://www.wikipedia.org/`

Figure 2.2: Life duration of non-surviving columns.

Having a deeper look at the life cycle of tables and columns reveals that they are either removed shortly after their birth or have a high chance to survive, which makes perfect sense since tables and columns that are around for a while are widely used by application code and therefore hard to remove. In the Wikimedia example, 12 tables are removed during the first 50 versions of their life—all other 34 tables survived the sampled evolution. Similar patterns can be observed for columns. Figure 2.2 shows the age of columns at the moment they are removed. Except of one outlier the 45 non-surviving columns have been removed within the first 80 versions after their birth; all other 240 columns survived. All the collected numbers match the findings of Curino et al. [40] and Vassiliadis et al. [113]. As the study of Vassiliadis covers more projects, there are interesting extensions to our observations: wide tables have a higher chance to survive, and surviving tables undergo many changes during their life time, while tables with less changes are more likely to be removed eventually.

In sum, it is essential to continuously evolve the database along with the application. We observed that the increasing complexity of the schemas induces a higher effort for developers to keep data and code consistent—hence, we conclude that developers urge for an improved support of database evolution to replace the error-prone and expensive manual solutions. Developers want evolve a database schema to a new version just as easy and agile as they can evolve application code. The **agile evolution** of production databases particularly includes the corresponding evolution of already existing data. In order to prevent from conflicts with other developers, users, and DBAs, all created schema versions should be available in the database at the same time. Developers want to test and further evolve new schema versions and also **manage the schema versions**, which might include to merge different schema versions or to drop unused versions. Users want to run applications in their preferred versions, so multiple **schema versions** have to **co-exist** in the database, but all should represent the same conceptual data set. Finally, DBAs want to **independently manage the physical materialization** of all the schema versions, since it is not acceptable that developers have to rewrite their applications or users cannot use their application any longer just because the DBA performs slight changes on the physical schema. In contrast, the current best practice is that big teams of developers and DBAs carefully implement all the kinds of evolution and migration between schema versions and execute them in big-bang migrations hoping that no data will be lost and all applications continue running, which consumes vast development resources and is highly error-prone [27]. While all the discussed features would improve the agile database development, the **ACID properties** of traditional relational databases are not negligible and have to be guaranteed in multi-schema-version database systems at any time. This particularly means that every single schema version behaves like a full-fledged single-schema database—a single application does not notice that its data is managed in a multi-schema-version database system.

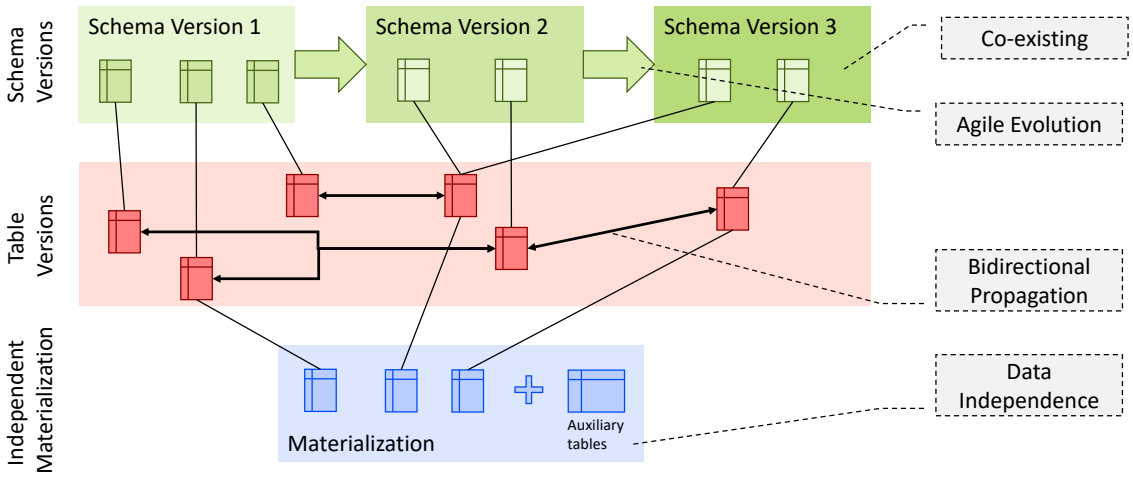


Figure 2.3: Envisioned multi-schema-version database system.

2.2 MULTI-SCHEMA-VERSION DATABASE SYSTEMS

Multi-schema-version database systems facilitate multi-schema-version data management. They manage multiple co-existing schema versions within one database to support the continuous evolution of the database schema and the underlying data as shown in Figure 2.3. Therefore, multi-schema-version database systems provide an evolution language that allows evolving an existing schema version to a new one just as fast as agile developers evolve an application. Each schema version consists of a set of **table versions** and the evolution language evolves these table versions in the given schema version to new table versions in the new schema version. Multi-schema-version database systems store the evolution of the table versions explicitly within the database catalog. A **schema version** is a subset of all table versions in this catalog—table versions that do not evolve between schema versions are shared among those schema versions.

Multi-schema-version database systems expose the data through multiple schema versions, while each single schema version can be accessed by applications just like a regular single-schema database. Hence, multi-schema-version database systems ensure that the transaction and consistency guarantees still hold for each single schema version. After developers create a new schema version in the multi-schema-version database system, this new schema version becomes immediately available to the applications and **co-exists** with all other schema versions. Data written in any schema version is correctly visible when reading the respective schema version again; further the write operation is propagated to all other schema versions along the schema version history in a best-effort manner. To facilitate the meaningful propagation of data between all schema versions both forwards and backwards, the evolution language needs to be **bidirectional**.

Having multiple schema versions upon one data set naturally raises the challenge to find a good physical materialization that serves all schema versions and provides the best overall performance for a given—potentially changing—workload. Having a materialization tailored to the current workload and adjusted when the workload mix changes, calls for full **data independence**: Multiple logical schema versions that are accessible for the applications represent the same conceptual data set (logical data independence) and the physical materialization of this conceptual data set can be freely changed (physical data independence) without limiting accessibility of any logical schema version. Hence, the logical and physical data independence for databases defined within the ANSI-SPARC architecture [1] have to be provided by a multi-schema-version database systems, as well.

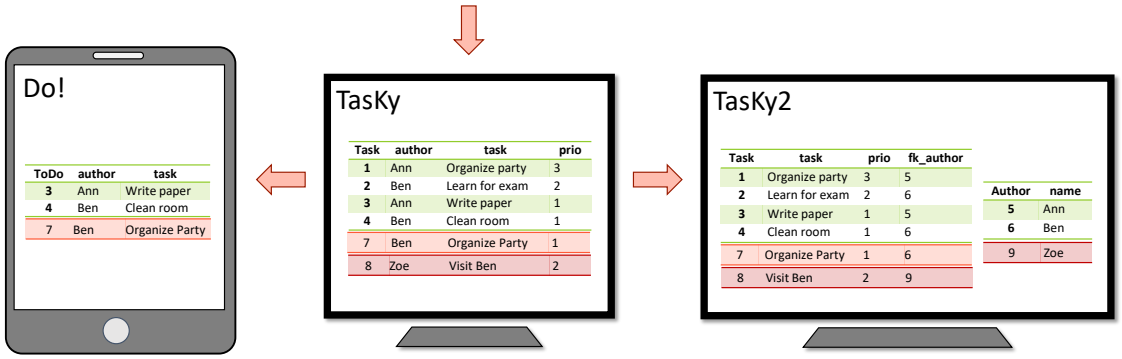


Figure 2.4: The exemplary evolution of TasKy.

Multi-schema-version database systems need only a subset of the table versions to be materialized. Generally speaking, read operations are fastest when the accessed table versions are materialized. The further away the next materialized table version, the higher the overhead for reading. The obvious solution of materializing every single table version implies a high overhead for writing and may hit memory limits, hence an adviser tool to solve the challenging **optimization problem to find the best materialization** for the current workload could be a helpful extension for multi-schema-version database systems. Since a chosen subset of all table versions does not necessarily cover the whole conceptual data set, the multi-schema-version database system also identifies and persists the otherwise lost information; this **auxiliary information** is persisted in auxiliary tables along the materialized table versions. The multi-schema-version database system needs to guarantee that the managed auxiliary tables cover the whole conceptual data set and no data is lost. Putting it all together, multi-schema-version database systems allow creating, managing, and accessing multiple logical schema versions upon one shared conceptual data set with an independent physical materialization.

2.3 USER STORY – THE TASKY EXAMPLE

InVerDa is a multi-schema-version database system developed in this thesis. We will use the running example of a task management software called TasKy (Figure 2.4) to illustrate both the requirements and our solution InVerDa. TasKy is a desktop application which is backed by a central database. It allows users to create new tasks as well as to list, update, and delete them. Each task has an author and a priority between 1 and 3. Tasks with priority 1 are the most urgent ones. The first release of TasKy stores all its data in a single table `Task(author,task,prio)`. At this point, there is only one existing schema version, hence it matches exactly the physical table schema without any delta code. TasKy has productive go-live and users begin to feed the database with their tasks.

Creating new schema versions: TasKy gets widely accepted and after some weeks it is extended by a third-party phone app called Do! to allow users accessing their most urgent tasks anywhere with their mobile. However, Do! expects a different database schema than TasKy is using. The Do! schema consists of a table `Todo(author,task)` containing only tasks of priority 1. Obviously, the initial schema version needs to stay alive for TasKy, which is broadly installed. Traditionally, we would use a view to create an external schema fitting Do!. Since views are not necessarily updatable this likely also includes writing triggers for the propagation of writes in Do! back to what TasKy sees in the database. InVerDa greatly simplifies the job as it handles all the necessary delta code for the developer. The developer merely executes the following BiDEL script to create the new schema version Do!:

```
1: CREATE CO-EXISTING SCHEMA VERSION Do! FROM TasKy WITH
2:     PARTITION TABLE Task INTO Todo WITH "prio=1";
3:     DROP COLUMN prio FROM Todo DEFAULT "1";
```

It instructs InVerDa to derive a new schema version Do! from the schema version TasKy by creating a horizontal partition of `Task` with `prio=1` and dropping the priority column. Executing the script creates a new schema including the view `Todo` as well as delta code for propagating data changes. When a user inserts a new entry into `Todo`, this will automatically insert a corresponding task with priority 1 to `Task` in TasKy. Equally, updates and deletions are propagated back to the TasKy schema. Hence, the TasKy data is immediately available to be read and written through the newly incorporated Do! app by simply executing the above BiDEL evolution script. At this point InVerDa has already simplified the developer's job significantly.

Rolling upgrade: Assume the TasKy application is further refined to make it more useful for its users. For the next release TasKy2, it is decided to normalize the table `Task` and store the authors in a separate `Author` table. For a stepwise roll-out of TasKy2, the old schema of TasKy has to remain alive until all clients have been updated. Again, InVerDa does the job after the developer executes the following BiDEL script:

```
1: CREATE CO-EXISTING SCHEMA VERSION TasKy2 FROM TasKy WITH
2:     DECOMPOSE TABLE Task INTO Task(task,prio), Author(author) ON FK fk_author;
3:     RENAME COLUMN author IN Task TO name;
```

InVerDa creates the new schema version TasKy2 and decomposes the table version `Task` to separate the tasks from their authors while creating a foreign key, called `fk_author`, to maintain the dependency. Additionally, the column `author` is renamed to `name`. InVerDa generates delta code to make the TasKy2 schema immediately available. Write operations to any of the three schema versions are propagated to all other schema versions. Figure 2.4 shows two examples: First, Ben uses the mobile Do! app to note that he will organize a party. This todo item is also visible in the two desktop versions with the priority 1. Second, Zoe uses the initial TasKy application to note down that she will visit Ben. When she moves to the new TasKy2 application this entry will be visible as well; Zoe is created as a new author and linked to the added task using the foreign key `fk_author`. Further, assume user Ann has already upgraded to TasKy2 and changes the priority of *Organize party* to 1, then this task will immediately occur in the Do! app on her phone. After the party, Ann deletes this entry using Do!, which also removes this task from the other schema versions. InVerDa generated three co-existing schema versions for us and any write operation on any schema version is propagated to all other versions. The developer merely defines the evolution between the schema versions with BiDEL. BiDEL's SMOs carry enough information to generate all the required delta code automatically without any further interaction of the developer.

Schema version management: Having multiple schema versions within one database calls for means to manage them by e.g. merging or dropping them. After the successful roll-out, all users are now using either the new desktop application TasKy2 or the mobile application Do!, so we can simply drop the former schema version TasKy from the database with a single line of code:

```
1: DROP SCHEMA VERSION TasKy;
```

As a result, the schema version TasKy cannot be accessed or evolved any longer and InVerDa can use this knowledge to generate more efficient delta code.

Assume the developers of the Do! application want to co-evolve their schema with the evolution from the original TasKy to the new TasKy2. Traditionally, they would sit down, analyze the changes made from TasKy to TasKy2, try to apply the same changes to the Do! schema version, and finally derive the new co-evolved schema version DoTwo!. This process is cumbersome and error-prone and calls for

convenient automation—however, using traditional SQL for database evolution hides the intention of the developers between the lines of SQL-DDL and SQL-DML statements and makes automation of the co-evolution process very hard. In contrast, BiDEL's SMOs capture the developers' intentions and facilitate automation in the first place. InVerDa provides the tool miX, which allows developers to simply execute the following statement to derive the co-evolved schema version DoTwo!:

```
1: CREATE CO-EXISTING SCHEMA VERSION DoTwo! FROM Do! AND TasKy2;
```

Upon this statement InVerDa derives and executes the following sequence of BiDEL SMOs:

```
1: CREATE CO-EXISTING SCHEMA VERSION DoTwo! FROM Do! WITH
2:     DECOMPOSE TABLE Todo INTO Todo(task), Author(author) ON FK fk_author;
3:     RENAME COLUMN author IN Todo TO name;
```

The new schema version DoTwo! only contains those tasks with the highest priority and stores them separated from their authors. The co-evolution is semi-automatic—if there are ambiguities, InVerDa's miX asks the developers—however, in our scenario the versions can be merged deterministically. In sum, InVerDa greatly simplifies a database developer's life. It facilitates convenient SMO-based evolution from existing to new schema versions that all co-exist within the same database. InVerDa also supports the developers to manage the multitude of schema versions and e.g. drop them or merge them with miX. In the evaluation, we show that this saves lines of code in the orders of magnitude—hence InVerDa makes database evolution more robust and more agile.

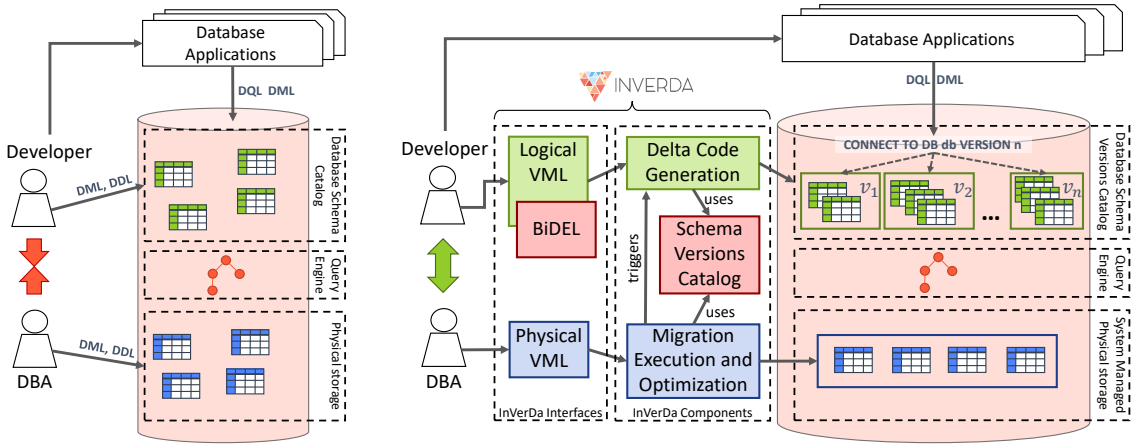
Physical data migrating: The data is primarily stored in a physical table matching the `Task` table of the initial schema version TasKy. All other schema versions are implemented with the help of delta code. The delta code introduces an overhead on read and write accesses to new schema versions. The more SMOs are between schema versions, the more delta code is involved and the higher is the overhead. In the case of the task management system, the schema versions TasKy2, Do!, and DoTwo! have delta code towards the physical table `Task`. Since, the initial schema version TasKy has already been dropped from the database and the majority of all users runs the new desktop application TasKy2 and the schema versions Do! and DoTwo! are still accessed but merely by a minority of users, it seems appropriate to migrate data physically to the table versions of the TasKy2 schema, now. Traditionally, developers would have to write a migration script, which moves the data and implements new delta code for all schema versions that have to stay accessible. All that can accumulate to some hundred or thousand lines of code, which need to be tested intensively in order to prevent it from messing up our data. With InVerDa, this task can be solved with a single line:

```
1: MATERIALIZE TasKy2;
```

Upon this statement, InVerDa transparently runs the physical data migration to schema TasKy2, while maintaining transaction guarantees and updating the involved delta code of all schema versions. No developers need to be involved. Since BiDEL's SMOs are bidirectional, hence agnostic to the materialization, it is possible to generate the new delta code without any further interaction of any developer. All schema versions stay available; read and write operations are merely propagated to a different set of physical tables after the migration.

Adviser for physical migration: InVerDa is not limited to physically migrating to one chosen schema version. In fact, we can instruct InVerDa to materialize practically any set of table versions, which are e.g. in different schema versions or even in-between schema versions. While this can include redundancy to speed up read performance, InVerDa validates that no data will be lost. On this more fine grained level, there is a huge space of possible physical table schemas, which is hard to understand and utilize for DBA. Therefore, InVerDa comes with an adviser that takes the current workload mix across all schema versions and proposes an optimized set of table versions to store physically. With the click of a button, the DBA execute the respective physical migration.

In sum, InVerDa allows the user to continuously use all schema versions, it allows the developer to continuously develop the applications, and it allows the DBA to independently adapt the physical table schema to the current workload or even let InVerDa do this job.



(a) Current architecture (b) Envisioned architecture

Figure 2.5: Architecture of current and envisioned DBMS.

2.4 ARCHITECTURE OF INVERDA

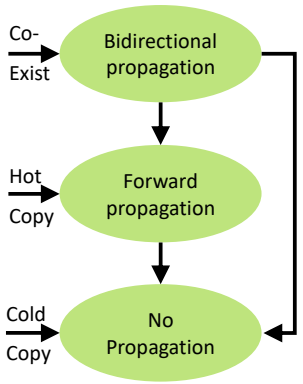
The requirements of multi-schema-version database systems are not satisfied by current DBMSes. Figure 2.5(a) depicts the rough architecture of a state-of-the-art relational DBMS, which we will analyze to propose an extended architecture for the management of multiple schema versions. Currently, database developers use SQL-DDL statements to define the database's schema. Applications use SQL-DQL and SQL-DML statements to read and write data within this specified schema. The schema catalog maps database objects from given statements to the actual physical storage and the query execution engine determines an efficient access plan for executing the query on the physical tables in order to respond fast and correctly. Whenever the database developers evolve the database, SQL-DDL is used to evolve the schema and SQL-DML to evolve the currently existing data accordingly. Whenever, the DBA wants to change the physical table schema, again SQL-DDL statements are used to change the schema and SQL-DML statements to migrate the data accordingly. This traditional approach separates the evolution of the schema (SQL-DDL) from the evolution of the data (SQL-DML); the intention of the developers or of the database administrator is lost between the lines and whenever one of them evolves the database, the work of the respective other gets most likely corrupted or invalidated. With this state-of-the-art setup, creating and managing multiple co-existing schema versions or optimizing the physical table schema becomes an error-prone and expensive challenge.

Multi-schema-version database systems, such as InVerDa, let the database developers and the DBA work independently on their respective tasks and provide robust database evolution support for all of them. The database developers create and manage table versions in logical schema versions, while the DBA can independently migrate the database physically to a set of potentially redundant table versions with the click of a button. This is not supported by current DBMSes, as their architecture is restricted to one schema version, which is directly mapped to physical tables in the storage. The developers and the DBA work on the same database objects and cannot evolve them independently.

InVerDa's architecture is shown in Figure 2.5(b). InVerDa simply builds upon the existing relational DBMSes and adds schema evolution functionality to support multiple schema versions at the same time. InVerDa's functionality is exposed via a Version Management Language (VML) which is two-layered: (1) the logical VML (lVML, Figure 2.6) allows developers to create multiple schema versions and manage e.g. the synchronization between them and (2) the physical VML (pVML, Figure 2.7) allows the DBA to physically manage the materialization for all co-existing schema versions within the database. The lVML and the pVML span a space of possible states a multi-schema-version database system can have. In the following we discuss the two dimensions of this state space in more detail.

```
1: CREATE (CO-EXISTING | HOT COPY | COLD COPY)
          SCHEMA VERSION name_new [FROM name_old]
          WITH SMO_1; ... SMO_n;
2: CREATE (CO-EXISTING | HOT COPY | COLD COPY)
          SCHEMA VERSION name_new FROM name_1 AND name_2;
3: ALTER SCHEMA VERSION name TO (HOT COPY | COLD COPY);
4: DROP SCHEMA VERSION version_n;
```

(a) lVML syntax



(b) lVML state chart

Figure 2.6: Logical Version Management Language (lVML).

lVML: The lVML statements as shown in Figure 2.6(a) allow to create, evolve, and drop logical schema versions in the multi-schema-version database system as well as to set the degree of synchronization between all managed logical schema versions. For the creation and evolution of schema versions, the lVML statement (Line 1 in Figure 2.6(a)) takes a sequence of SMOs from a DEL called BiDEL. When creating a new schema version as an evolution from an existing schema version, developers need to specify the degree of synchronization between the old and the new schema version. There are three options and possible transitions between them as can be seen in Figure 2.6(b): (1) **co-existing** schema versions working on the same shared data set with write operations being propagated both forwards and backwards along the schema version history, (2) a **hot copy** with changes being propagated forwards from the old schema version to the new version but not vice versa, and (3) a **cold copy** with the new schema version being merely a copy of the old version's data at evolution time but no write operations are propagated in any direction at any time. Developers have to set the degree of synchronization together with the actual evolution to the new schema version but can also loosen it subsequently.

When the developers choose the hot or the cold copy, the new schema version becomes available as soon the data is physically copied to this new schema version. When the developers choose the co-existing schema versions option, the new schema version becomes immediately available and co-exists with all other schema versions—the data remains physically in its current state and the new schema version is merely created virtually. Write operations in any schema version are immediately reflected in the other schema versions as well. For data access in multi-schema-version database systems, the applications simply define the schema version to access when connecting to the database and can then use it like a full-fledged single-schema database.

Subsequently, the developers can alter the degree of logical synchronization (Line 3 in Figure 2.6(a)) by specifying the target schema version of the respective evolution and the intended propagation strategy. Please note that the degree of synchronization can only be reduced (cf. Figure 2.6); whenever the synchronization has been broken in one direction, we cannot guarantee consistency across schema versions any longer, so it is e.g. not possible to safely return to co-existing schema versions after the two schema versions have been independently updated cold copies. Of course, a reconciliation strategy is imaginable but not trivial: Assume that updates to the tasks in our TasKy example (Figure 2.4 on page 28) are not propagated between the schema versions and the priority of Ann's first task has been updated from 3 to 2 in TasKy but to 1 in TasKy2. When reinforcing bidirectional propagation for co-existing schema versions, naïvely we can either decide on one value that is used in both schema versions or add the ambiguity to the auxiliary information. While the first solution breaks the data independence, the latter makes any subsequent write operation ambiguous as well. The future research targeting at the reconciliation problem is promising but out of the scope of this thesis.

The lVML statement to evolve schema versions (Line 1 in Figure 2.6(a)) takes a sequence of BiDEL SMOs to describe this evolution and define the data propagation between the created schema versions in a multi-schema-version database system. BiDEL provides a comprehensive set of bidirectional SMOs that evolve up to two *source table versions* to up to two *target table versions*. Each table version is created by exactly one *incoming SMO* and can be further evolved by arbitrary many *outgoing SMOs*. The SMOs allow to create or drop or rename both tables and columns, and to split and merge tables both vertically and horizontally. Specifically, BiDEL offers the following SMOs to describe the evolution of a database schema version:

- `CREATE TABLE`, `RENAME TABLE`, `DROP TABLE`
- `ADD COLUMN`, `RENAME COLUMN`, `DROP COLUMN`
- `DECOMPOSE`, `JOIN`
- `PARTITION`, `MERGE`

The first six operations are generally known from standard SQL-DDL and follow the same semantics. The `DECOMPOSE` SMO vertically splits a source table version by distributing its columns into two new target table versions—columns can also occur in both or in no target table version. To preserve the link between split tuples, the `DECOMPOSE` SMO also provides to generate a new foreign key between the two resulting tables. Its inverse SMO is the `JOIN` SMO as known from standard SQL-DQL. Further, the `PARTITION` SMO moves the tuples of the source table version to two new table versions according to specified selection criteria—the criteria may also overlap and do not necessarily cover all tuples of the source table. Its inverse SMO is the `MERGE` SMO, which merely unites the tuples of two source table versions into a new target table version.

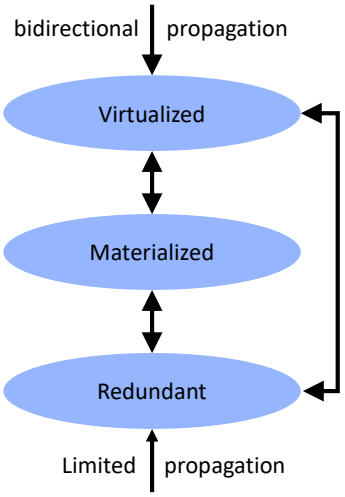
We focus the considered expressiveness of BiDEL on the evolution of tables with the relational algebra; the evolution of further artifacts like constraints and functions is promising future work [37]. Our DEL BiDEL is relationally complete—it allows to specify any forward evolution which can be expressed with a relational algebra expression—as we will formally evaluate in this thesis.

Another unique feature of BiDEL is its bidirectionality. Whenever, developers need backward propagation of data e.g. for co-existing schema versions, the developers have to provide additional arguments to specify the backward propagation of data. These arguments are optional since backward propagation is not required for hot and cold copies, but they are mandatory for bidirectional propagation. Bidirectional SMOs are the essential prerequisite for co-existing schema versions within one database. The arguments of each BiDEL SMO gather enough information to facilitate the automatic generation of delta code for the full propagation of reads and writes between schema versions in both directions. For instance, `DROP COLUMN` requires a function f that computes the value for the dropped column if a tuple, inserted in the new schema version, is propagated back to an old schema version, where the dropped column still exists.

The lVML does not only allow creating new schema versions but also to merge two existing schema versions semi-automatically—e.g. two branches developed by different teams (Line 2 in Figure 2.6(a)). For this purpose, InVerDa contains the miX tool. Thanks to the SMO-specified evolutions, the developers' intention is kept explicitly which facilitates miX to semi-automatically merge the intentions of two different evolutions in the first place. Further, schema versions can be dropped if not used any longer, which drops the schema version itself but maintains the table versions if their data is still needed in other versions (Line 4 in Figure 2.6(a)).

```
1: MATERIALIZE {name_version}+;
2: MATERIALIZE {(name_table, revision_table)}+;
3: MATERIALIZE AUTOMATICALLY [LIMIT bound];
```

(a) pVML syntax



(b) pVML state chart

Figure 2.7: Physical Version Management Language (pVML).

pVML: InVerDa's pVML commands allow the DBA to change the physical data representation. Co-existing schema versions, which do bidirectional propagation of data, maintain one conceptual data set and the DBA can choose its materialization. The syntax of the pVML is shown in Figure 2.7(a); basically the DBA can manually choose to materialize a set of schema versions (Line 1 in Figure 2.7(a)) or a set of table versions (Line 2 in Figure 2.7(a)) or the DBA can instruct InVerDa to automatically determine the best materialization for the current workload and apply it (Line 3 in Figure 2.7(a)). From the view point of a single evolution between table versions or schema versions, the data can be primarily stored in the source, or in the target, or in both versions; the respective state space and the pVML transitions are shown in Figure 2.7(b).

By default data is materialized only in the source schema version. Assuming a table is partitioned in a schema evolution step, the data remains physically unpartitioned—the evolution is called *virtualized*. With a migration command the physical data representation of this evolution can be changed so that the data is also physically partitioned—the evolution is called *materialized* then. Further, InVerDa allows redundant materialization, so the data can be stored both unpartitioned at the source side and partitioned at the target side—the evolution is called *redundantly materialized* then. Since not all evolutions are information preserving, InVerDa uses *auxiliary tables* that store the otherwise lost information. These auxiliary tables are initially empty but fed with information whenever needed to not lose data in the virtualized schema version—e.g. the values of an added column.

Please note that InVerDa is not limited to the materialization of one or two concrete schema versions. In multi-schema-version database systems, it is feasible to store the data non-redundantly or partial redundantly in slices of the evolution history and all other versions are virtually provided based on delta code and auxiliary tables. Basically, we can materialize a subset of all table versions, which spans a very large solution space of possible materialization. Given the set of table versions to materialize, InVerDa automatically adds further table versions that are required to cover the whole data set at least once. Assume, we materialize one schema version with one table, but another schema version has an additional second table that was created by a `CREATE TABLE` SMO, InVerDa has to materialize this table version as well. For each table version, a valid materialization ensures that there is a propagation path through the SMOs that ends up at materialized table versions. Internally, InVerDa translates the set of materialized table versions to materialization states of SMOs: for each SMO there is one direction—forwards or backwards—that leads to the closest materialized table versions.

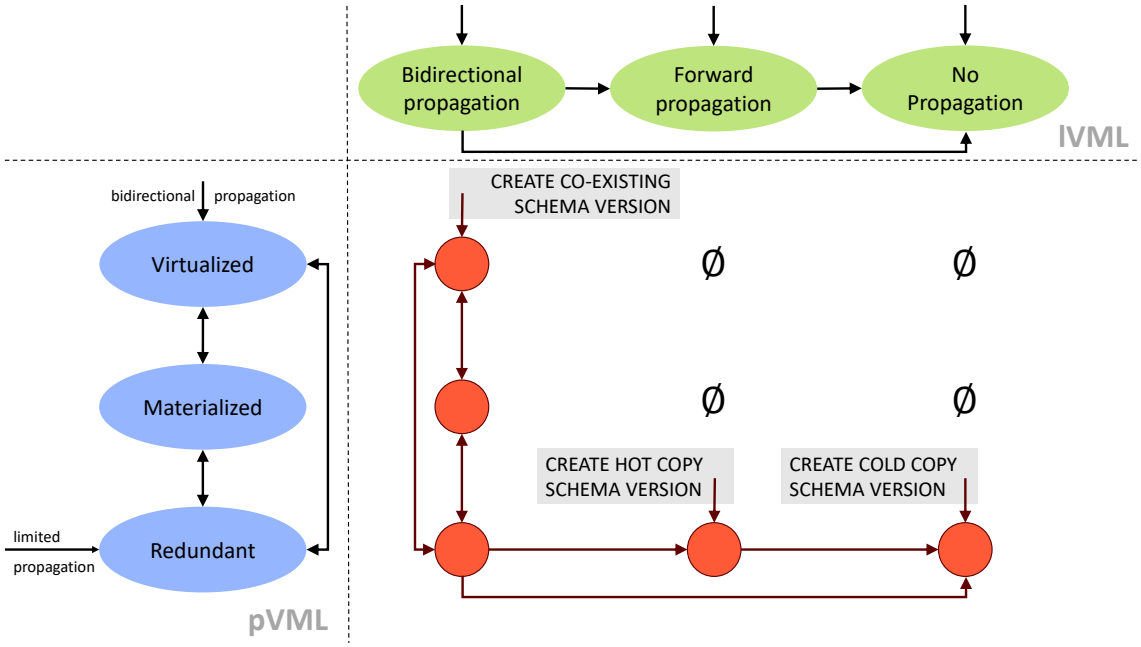


Figure 2.8: Evolution states.

Further, the pVML allows the DBA to instruct InVerDa to propose an optimized materialization for the current workload, which calls the adviser that takes the current workload and the schema versions catalog as input and returns a feasible materialization. The DBA can confirm the proposed materialization and instruct InVerDa to physically migrate the database accordingly. Without any further interaction of the developers, the physical table schema is optimized for the current workload and all schema versions continuously co-exists.

The full data independence is exclusively realized for co-existing schema versions with bidirectional data propagation between these schema versions. In contrast, both the hot and the cold copy separate the data sets of the two schema versions and do not guarantee any consistencies between them. Thus, data is replicated at evolution time and updated (partly) independent afterwards. There is no freedom in the materialization, but we materialize every copy individually. Obviously, we could do this lazily by storing merely the delta when data is changed in the non-physical version; however, this is a typical data versioning problem in temporal databases with sophisticated solutions in the literature [119, 80, 120] that are orthogonal to our work and thereby out of scope.

Figure 2.8 illustrates possible combination of the discussed logical states (Figure 2.6(b)) and physical states (Figure 2.7(b)) for the evolution from one source to one target version. With the pVML, the DBA has the freedom to change the physical table schema given the developers decided for co-existing schema versions. Whenever, the developers use lVML statements to switch to a limited propagation between the schema versions (hot or cold copy), we first need to ensure that data is replicated to both schema versions and this cannot be undone ever after. In sum, the DBA uses pVML to manage the physical table schema independently of the logical schema version management done by the database developers with lVML—so InVerDa realizes full **data independence** in the context of multi-schema-version database systems. It allows to physically migrate the database to one or multiple redundant table versions. When running such a physical migration, the availability of the logically co-existing schema versions is not affected at any time and all involved delta code is automatically generated from the given SMOs without any developer being involved.

System integration: In our prototypical implementation of InVerDa[5], we create the co-existing schema versions with views and triggers in a common relational DBMS. InVerDa interacts merely over common DDL and DML statements, data is stored in regular tables, and database applications use the standard query engine of the DBMS. To process data accesses of database applications, only the generated views and triggers are used and no InVerDa components are involved. The employed triggers can lead to a cascaded execution, but in a controlled manner as there are no cycles in the version history. InVerDa's components are only active during database development and migrations. Thanks to this architecture, InVerDa easily utilizes existing DBMS components such as physical data storage, indexing, transaction handling, query processing, etc. without reinventing the wheel. Please note that InVerDa is by no means limited to generating views and triggers. Alternatively, the concepts of InVerDa could also be used to generate delta code e.g. within the application or as optimized execution planes within the database.

As can be seen in Figure 2.5(b) on page 31, InVerDa adds three components to the DBMS to facilitate multi-schema-version data management: (1) The *delta code generation* creates views and triggers to expose schema versions based on the current physical data representation. Delta code generation is either triggered by developers issuing lVML commands to create a new schema version or by the DBA issuing pVML to change the physical data representation. The delta code itself are standard commands of the DBMS's query engine. (2) The *migration execution and optimization* orchestrates the actual migration of data from one physical representation to another and the adaptation of the delta code. It also contains the adviser for the optimized physical materialization for the current workload. The data migration is done with the help of query engine capabilities as well. (3) The *schema versions catalog* maintains the *genealogy of schema versions* and is the central knowledge base for all schema versions and the evolution between them. To this end, the catalog stores the genealogy of schema versions by means of a directed acyclic hypergraph (T, E). Each vertex $t \in T$ represents a table version. Each hyperedge $e \in E$ represents one SMO instance, i.e., one table evolution step. An SMO instance $e = (S, T)$ evolves a set of source table versions S into a set of target table versions T. Additionally, the schema versions catalog stores for every SMO instance the SMO type (decompose, merge, etc.), the parameter set, and its state of materialization. The schema versions catalog maintains references to tables in the physical storage that hold the payload data and to auxiliary tables that hold otherwise lost information of the not necessarily information preserving SMOs. Each schema version is a subset of all table versions in the system. At evolution time, InVerDa uses this information to generate delta code that makes all schema versions accessible. At query time, the generated delta code itself is executed by the existing DBMS's query engine—outside InVerDa components.

To sum up, we describe the process for multi-schema-version data management with InVerDa: When developers execute lVML scripts with BiDEL-specified evolutions, the respective SMO instances and table versions are registered in the schema versions catalog. The materialization states of the SMOs, which can be changed by the DBA through migration commands, determine which data tables and auxiliary tables are physically present or not. InVerDa uses the schema versions catalog to generate delta code for new schema versions or for changed physical table schemas. The materialization state determines the way InVerDa generates delta code for each specific SMO: data is either propagated forwards (SMO is virtualized) or backwards (SMO is materialized). Data accesses of applications are processed by the generated delta code within the DBMS's query engine—no InVerDa components but only standard DBMS artifacts like the generated delta code (views and triggers) are involved to process data accesses. When developers drop an old schema version that is not used any longer, the schema version is removed from the catalog. However, the respective SMOs are only removed from the schema versions catalog as well if they are no longer part of any evolution that connects two remaining schema versions.

[5] An interactive online demonstrator is available at `www.inverda.de`

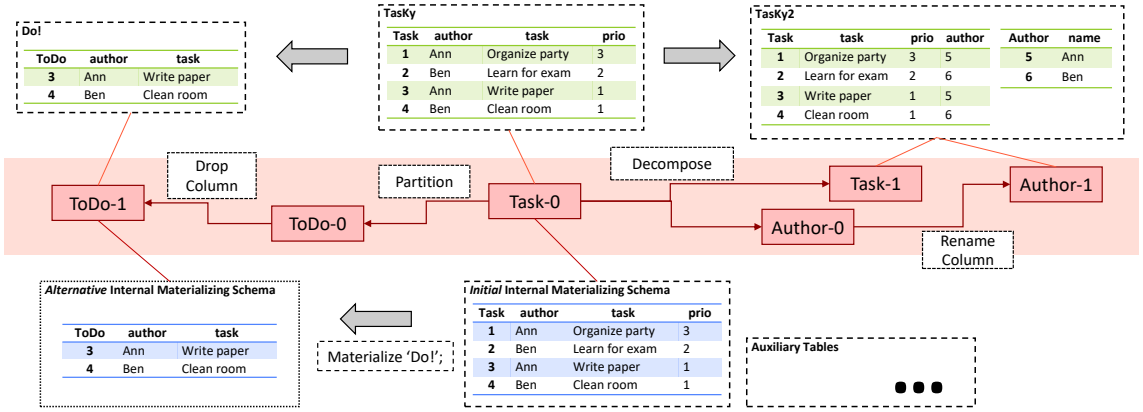


Figure 2.9: InVerDa architecture with the TasKy example.

Example: Let us revisit the TasKy example to discuss InVerDa's three-layered architecture as shown in Figure 2.9. On the upper logical layer, we have all the three schema versions TasKy, TasKy2, and Do!. InVerDa creates a separate schema for each version containing updatable views that act like tables in a standard database schema. On the lower physical layer, a subset of the table versions is physically stored as standard tables in the database. Finally, the conceptual layer interconnects the materialized table versions with the table versions in the logical schema versions. This conceptual layer is a representation of the schema versions catalog containing all table versions as well as all SMO instances. Any incoming read and write operations are propagated from the respective logical schema version on which they are issued through the SMOs to the materialized tables. Hence, all schema versions can co-exist in the multi-schema-version database system and write operation in any schema version are propagated to the materialized tables plus auxiliary tables making the written data immediately visible in all other schema versions as well.

Initially, the `Task-0` table is exclusively materialized, which makes any access to the TasKy schema version trivial. In contrast, accessing e.g. the Do! schema version requires to propagate both read and write operations through the DROP COLUMN and the PARTITION SMOs back to the physical table, which implies an overhead. Assume 99% of the users use Do!, we can easily increase the overall performance by moving the physical materialization to the Do! schema version with a single-line pVML statement. Now, any data access in the Do! schema version can be directly mapped to the materialized tables which significantly speeds up query processing (4 times faster in our evaluation). Users that access data in the schema versions TasKy and especially TasKy2 now face a higher overhead for the additional data propagation. Please note that the bidirectional semantics of BiDEL's SMOs also allow redundant materialization, so we could e.g. materialize both schema version Do! and TasKy2 to obtain a high read performance in both schema versions. Hence, InVerDa gives us the freedom to easily adjust the set of materialized table versions to a changing workload: true data independence.

In sum, the proposed architecture for a multi-schema-version database system satisfies the requirements that we derived from the analysis of software and database evolution in Section 2.1. Using the intuitive evolution language BiDEL, developers can easily create new schema versions that are immediately available. The schema versions catalog keeps track of the evolution between schema versions and thereby allows to propagate any read or write access between the schema versions. The actual physical materialization can be freely chosen by the DBA and does not affect the availability of the co-existing schema versions, still, it might greatly affect the performance for accessing these schema versions. The ACID properties are guaranteed on every single schema version, so developers and applications can work on one schema version of the multi-schema-version database system just as on a regular single-schema database.

After reviewing the related work in Chapter 3, the scientific contribution of InVerDa will be detailed. In Chapter 4, we consider the problem of designing a bidirectional evolution language to create and evolve new schema versions in a multi-schema-version database system. In Chapter 5, we discuss how to merge existing schema version into a new one with miX. So, we present two means to create new schema versions in the multi-schema-version database system. Afterwards, we focus on the physical data management: in Chapter 6 we exploit the bidirectionality of BiDEL to achieve full data independence but still guarantee to not lose any data at any time. Since the space for possible non-redundant and redundant physical materializations is huge, we present a respective adviser in Chapter 7. In Chapter 8, we discover and discuss various application scenarios and promising future work for our multi-schema-version database system InVerDa.

RELATED WORK

Software systems continuously evolve and so does the database in their backend. The need for decent database evolution support is a pressing challenge ever since databases are used for persisting the data within software systems. 40 years of intensive research and development aimed at easier, faster, and more robust database evolution support both by external tools and directly within the database. We will first detail on the objectives and the most important challenges posed by users, developers and DBAs w.r.t. multi-schema-version database systems (Section 3.1). Afterwards, we present related work tackling these criteria both in the fields of software evolution (Section 3.2), and database evolution (Section 3.3). Finally, we summarize the related work by evaluating the defined objectives for all discussed approaches and in comparison to InVerDa (Section 3.4).

3.1 OBJECTIVES

We structure and discuss the objectives for multi-schema-version database systems in three major categories: First, the suitable support for database evolution and co-existing schema versions within one database. Second, the flexible materialization of co-existing schema versions to guarantee full data independence. And finally, the comfortable version management focusing on merging different schema versions. We consider the following objectives for multi-schema-version database systems:

1. **Database evolution support**
    - **Database Evolution Language (DEL):** While SQL-DDL and -DML carefully distinguish between the schema evolution and the corresponding data evolution, an SMO-based DEL couples the data evolution to the schema evolution. Thereby, the actual evolution step and its intention are explicitly defined and preserved.
    - **Practical completeness:** A DEL is only practically applicable, if it is powerful enough to express any evolution scenario at hand. Whenever developers have to fall back on traditional manual implementations, all the problems solved by a multi-schema-version database system arise again.
    - **Relational completeness:** While practical completeness is an empiric measure, relational completeness is a formal guarantee stating that the respective DEL is at least as powerful as the relational algebra.
    - **Co-Existing schema versions:** When schema versions co-exist in the same database, there are different extents of synchronization between them: While common unidirectional DELs allow only the *forward propagation* of data—merely reading in new schema versions and writing in old schema versions is supported—a *bidirectional* DEL allows propagating data both forwards and backwards, which facilitates both reading and writing at both the old and the new schema version.
2. **Full data independence for multi-schema-version database systems**
    - **Flexible materialization:** Co-existing schema versions managed in one database require a materialization that serves the current workload on all schema versions as good as possible, and is easily adaptable when the workload mix changes. Full data independence allows changing the materialization without affecting the availability of any schema version.
    - **Adviser for physical materialization:** Since the search space of physical materializations for a given schema version history and a given workload is huge and difficult to explore manually, an adviser proposing the optimal materialization is desirable.
    - **Formally evaluated correctness:** The correctness and consistency of the data in multi-schema-version database systems needs to be formally guaranteed, which requires every single schema version to verifiably behave like a full-fledged single-schema database.

3. **Schema version management**

    - **Co-evolution of schema versions:** After branching the evolution of a database schema—e.g. when different development teams can work independently—tool support is desired to merge these branches into one consolidated schema version.
    - **(Semi-) automatic co-evolution based on intentions:** Merging two schema versions can be based on almost any criteria available for the input models/schemas, like e.g. the structure, data distribution, phonetics, etc. However, the Holy Grail is to merge the actual intentions of the developers.

There is plenty of work in the literature for each of these aspects individually. To the best of our knowledge, we are the first to realize a multi-schema-version database system that covers all the discussed objectives: Using the SMO-based bidirectional DEL BiDEL, our system InVerDa facilitates easy management of created schema versions, including merging and dropping them, as well as letting them co-exist within one database with a flexible materialization.

3.2 SOFTWARE EVOLUTION

The whole research field about the proper database evolution naturally originates from advances in continuously evolving software systems. While software development is already very agile and dynamic, database developers still struggle to keep the pace since database evolution is barely supported with tools and methods these days. Therefore, we take a look into the software technology community to learn about the requirements and obtain inspirations from their solutions for fast and agile software development. Specifically, we discuss support for the continuous evolution of software systems (Section 3.2.1), tools for co-existing software versions (Section 3.2.2), and existing approaches for the co-evolution of different artifacts in a software system (Section 3.2.3). Figure 3.1 summarizes the schematic functionality of all discussed tools and concepts, which we discuss in Section 3.2.4.

3.2.1 Continuous Evolution

Until the turn of the millennium, software developers mainly relied on the waterfall model [21, 14]: After following the path from a detailed requirement analysis over big upfront modeling and planning all the way to implementation and testing, the final software product has been released to the customer. This methodology quickly became unfeasible, since requirements keep changing and cannot be fixed in the very beginning of a software project. As a solution, **Agile Software Development** jump starts with a first version of the envisioned product very early, which is immediately shipped to the customer [18, 7]. This way, customers can experience the software live under real conditions and give valuable feedback to the developers w.r.t. prioritization of new features, bug-fixes, or non-functional properties like performance and availability. The running software product is continuously improved in many small iterative steps.

The mantra of agile software development is “Evolution instead of Revolution”, so a running software system is continuously improved in many small iterations. This is e.g. supported by **Extreme Programming** [17], which is a structured programming method heavily relying on team work and continuous communication between all stakeholders. One common extent of living it in practice is pair programming [117], where teams of two developers work together at one screen producing high-quality code due to the interactive and constantly justified programming.

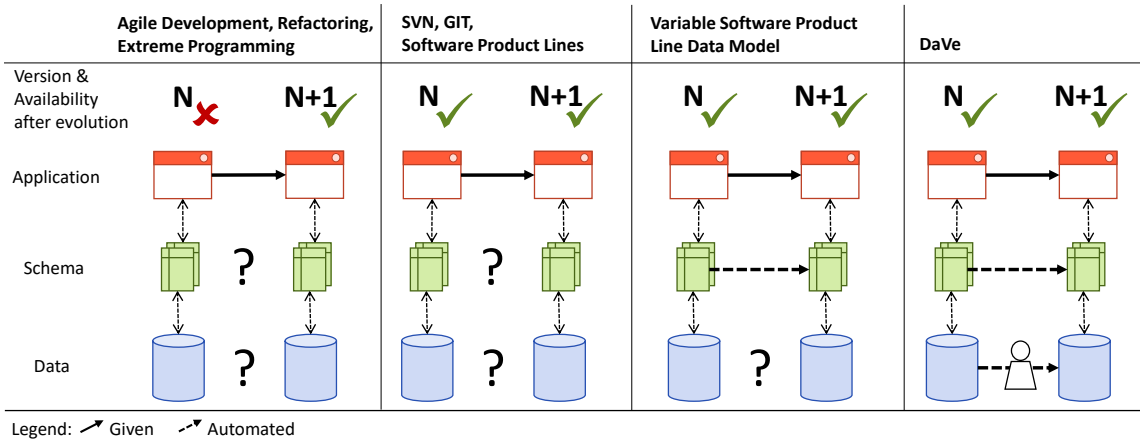


Figure 3.1: Related work in software development.

A central technique used to achieve this development speed is the use of **Refactorings** that allow changing a working software in a robust and controlled manner [51]. Multi-schema-version database systems adapt this principle. SMOs are refactorings on the database schema; their main difference to traditional SQL-DDL statements is their controlled and consistency preserving nature, as they also cover the evolution of the currently existing data significantly reducing the risk of ending up with an inconsistent database. In sum, the overall goal is to make database development just as agile as software development by changing the game from big upfront modeling to an incremental continuous evolution of the database.

3.2.2 Co-existing Versions

Also in software development multiple versions of the same software need to co-exist so different developers and different users can independently work on their preferred version of the software. For developers, the versioning of the code can be achieved with version control systems like SVN and GIT. For users, co-existing versions of an application can be generated and operated using e.g. software product lines.

Version control systems such as **Apache Subversion**[1], a.k.a. SVN, allow different developers to work independently on local copies of the same text or code files—concurrent changes are merged on the granularity of whole lines when developers commit their changes to the central repository—potential conflicts are either solved automatically on a syntactic level or manually by the developers if there are semantic conflicts. Further, different development teams can work on different global branches, so multiple versions of the same software can co-exist and can be developed independently. Afterwards, branches can be merged. Multi-schema-version database systems nicely complete this tooling for database developers. Even though SVN supports branching into different versions and merging them again, there are no consistency guarantees and the developers are heavily involved to branch and merge manually. SVN can merely merge SQL evolution scripts on a syntactical level, without any understanding of the underlying semantics, the currently existing data, and the developers' intentions. Multi-schema-version database systems fill this gap.

[1] `https://subversion.apache.org/`

Another widely used version control system for software developers is **GIT**[2]. GIT, provides the same functionality as SVN, plus an additional staging area, which decouples the local versioning from the global versioning. After implementing e.g. new features, developers commit their changes only locally into the staging area first. When committing from the staging area to the global repository, which is also visible to all other developers of the project, all locally performed changes can be reviewed and selectively pushed to the global repository. This allows more comfortable software development, since developers have the full functionality of a version control system both on a local level and on a global level. These two levels are decoupled as developers can precisely choose which new features should be pushed to the globally visible repository at which time.

Agile software development allows to easily adjust and extend applications exactly to the need of the users. However, the multitude of users can be very heterogeneous. All the users of a software product need something similar, e.g. a self-driving car, however, this may range from forklifts on company grounds, over trucks on the highway, all the way to fully automatic cars safely driving through the rush hours in cities. Most features overlap, so they should be implemented only once. Nevertheless, there are special features that are only used by a subset of the customers, which calls for one individual version of the software product per customer. It has to be avoided that, for a new customer, an existing product is replicated and then customized because of the single source principle. It is hardly possible to consistently fix potential bugs in each and every customer product individually [92]. As a consequence, the techniques and methodologies of **Software Product Line Engineering** [93] are applied in software development. Software product line engineering defines a family of closely related software systems consisting of common and variable functionality in order to manage variability. The variability is commonly described with feature models [73] wherein features are arranged in feature trees [31]. To generate a specific software product version from the software product line, the customer merely chooses the desired set of features, so different customers can have different versions of the same software at the same time.

Software product line engineering is primarily a concept that simplifies building customized products from the same source code. Once a product is deployed for a customer, there is no back link from the product to the product line: not for the code and especially not for data that is persistent in the deployed product. This makes the **database evolution in software product lines** a tough challenge. In general, the evolution of a database schema for the concrete software product of a customer is analyzed in [75]. Delta-Oriented Programming is used to add delta modules, defined by SQL-DDL statements, to a core module incrementally, based on the product configuration. Database constraints are generated for the delta scripts to ensure a valid global database schema. Modeling data variability in software product lines is typically based on feature modeling as used in software product line engineering [2]. To define variability of data concepts in the variable data model, the extended Feature Assembly Modeling Technique is used to specify persistence features in the feature model of the software product line [3]. However, this technique only considers the initial derivation of a product's database schema and not the continuous evolution that includes already existing data.

To the best of our knowledge, there is no published research on the concurrent evolution of both the schema and the currently existing data in software product lines. Though, this is a major challenge: Each selectable feature requires a shared or individual database schema to store its data—and this schema may evolve over time. Since there can be easily thousands of features, composing the database schema and schema evolutions for one generated product requires tremendous manual work in practice. These challenges have been discussed in detail and pragmatically solved by **DaVe** [58]. However, **DaVe** merely offers tool support for the manual evolution and migration of the database. An automated and controlled process for database evolution in software product lines is still missing.

[2] `https://git-scm.com/`

3.2.3 Co-evolution

In software development, the co-evolution of different artifacts in a software system has gathered a lot of attention, especially targeting the co-evolution of artifacts on different abstraction- or architecture-levels. Abstractly speaking, whenever a meta model evolves, the corresponding model instances need to be evolved as well [88]. The most commonly applied solutions to tackle this **Model Co-evolution** are transformation-based [114, 64], so the evolution is not derived after the fact from the old and the new version, but the transformation is explicitly stated by the developers and the new version is created based on the old version plus the given transformation. Especially, if an artifact is derived from a given meta model, any transformation on the meta model can be automatically applied to the artifact without having the developers in the loop [33].

Especially interesting from a database developer's point of view is the **co-evolution of the database schema and program code** accessing this database schema. Cleve et al. [34] present a system that supports the developers in modifying the database access code from a given database schema. Whenever the database schema is changed, which may include adding, deleting, or replacing any database object, the developers get a list of affected code snippets and can easily adapt them to the evolved schema. However, the supported evolution steps of the database schema do not cover the data at all. Terwilliger et al. go a step further and enable the automated co-evolution of both the conceptual model and the physical database and the mappings in between them [108, 109]. Their basic operation is to add classes within an inheritance hierarchy and the new concepts are immediately and automatically made available in the database. Multi-schema-version database systems provide an appealing alternative to the costly evolution propagation between the database schema and application code. Since former schema versions stay alive even though the database may be physically already migrated to a new schema version, there is no need to rewrite the application's data access code in the first place. Or in other words: For the evolution of database schemas, multi-schema-version database systems encapsulate the co-evolution of the data access code into the generated views that keep old schema versions accessible.

3.2.4 Summary

Figure 3.1 summarizes the schematic functionality of all discussed evolution support tools in software development. At first, we discussed agile development methods that are based e.g. on refactorings and extreme programming techniques, which facilitate a fast and robust forward evolution of the application itself, however, co-existing versions or the agile evolution of the database are not supported. Secondly, we presented techniques like code version control systems and software product lines that allow managing multiple co-existing versions of one software. Afterwards, we additionally incorporated the evolution of the database schema along with the agile evolution of the application within e.g. software product lines. The main takeaway message is that the evolution of the database schema along with the evolution of software products in a software product line is very challenging and is only supported for empty databases without any data. Hence, the initial schema for a software product can be generated, but potentially already existing data will be lost. The only system that provides tool support for the database evolution in software product lines is DaVe, which merely guides the developers through the manual evolution of the database and does not automate it.

In sum, the software technology community developed a wide bouquet of tools and methods for the agile evolution of software systems and the management of multiple versions of e.g. code and software products. Still, most approaches leave out the database evolution, so the discussed related work merely serves as a motivation and inspiration for agile database development and schema version management facilities.

3.3 DATABASE EVOLUTION

Relational databases are the most established and straight forward choice to manage and persist the data of a software system. This places the relational databases in the center of continuously evolving software systems, motivating a multitude of research and engineering literature regarding the database evolution. We will not survey the hundreds and thousands of papers here but just focus on those aspects that are relevant for multi-schema-version database systems. For the interested reader, Roddick published a bibliography of database evolution literature back in 1992, which is still a great starting point into the subject [98]. More recently in 2006, Rahm et al. launched an online bibliography that is regularly maintained [97] and Hartung et al. published a survey focusing on database evolution in both research projects and commercial DBMSes in 2011 [57]. Database evolution is also a challenge in related fields, e.g. Manousis et al. surveyed database evolution in modern data warehouses [87].

Multi-schema-version data management does not exclusively focus on the plain forwards evolution of the database but uses bidirectional evolutions to continuously serve both old and new schema versions. However, we did not think up this on ourselves, but a lot of database evolution research moved forwards to incorporate backward evolution—hence, bidirectional evolution. Terwilliger et al. conducted a survey motivating further work towards the bidirectional database evolution by collecting and comparing different approaches like SMO-based evolutions, lenses, channels, etc. [110], which we will discuss in detail in this section. To underline the importance but also the outreach of bidirectional transformation, there was even a Dagstuhl-Seminar held in 2011, where database researchers met with colleagues from both software engineering and programming language and graph transformations for interdisciplinary exchange regarding both previous findings and the future research focus of bidirectional transformations [67].

In this section, we will go through our defined objectives in detail. First, we discuss the evolution support presented in related works with focus on co-existing schema versions in Section 3.3.1—the presented approaches are comparatively illustrated in Figure 3.2. Second, we analyze related work targeting data independence in the context of multi-schema-version database systems in Section 3.3.2 with the approaches being illustrated in Figure 3.3. Third, the co-evolution of different schema versions is discussed in Section 3.3.3 and illustrated in Figure 3.4. Finally, we comparatively discuss and summarize all presented approaches in Section 3.3.4.

3.3.1 Database Evolution Support

Continuous evolution: Scott Ambler [8] approaches database evolution from the practical side by proposing more than 100 **Database Refactorings** ranging from simple renamings over changing constraints all the way to exchanging algorithms and refactoring single columns to lookup tables[3]. One essential characteristic of these refactorings is that most of them couple the evolution of both the schema and the data in consistent and easy to use operations—just as SMO-based DELs do.

The most advanced **SMO-based DEL** we are aware of, is **PRISM** by Carlo Curino et al. that includes eleven SMOs [38]. The proposed SMOs are very intuitive and easy to learn, also because they evolve a maximum of two source tables to a maximum of two target tables. Specifying a database evolution with SMOs opens a whole new field of interesting research challenges, which we just started exploring. A milestone in this process was the introduction of **PRISM++**, which extends the DEL to also include Integrity Constraint Modification Operations (ICMOs) that allow to create and drop both value

[3] List of refactorings is freely available: `http://databaserefactoring.com/` (Accessed on 30.08.2017)

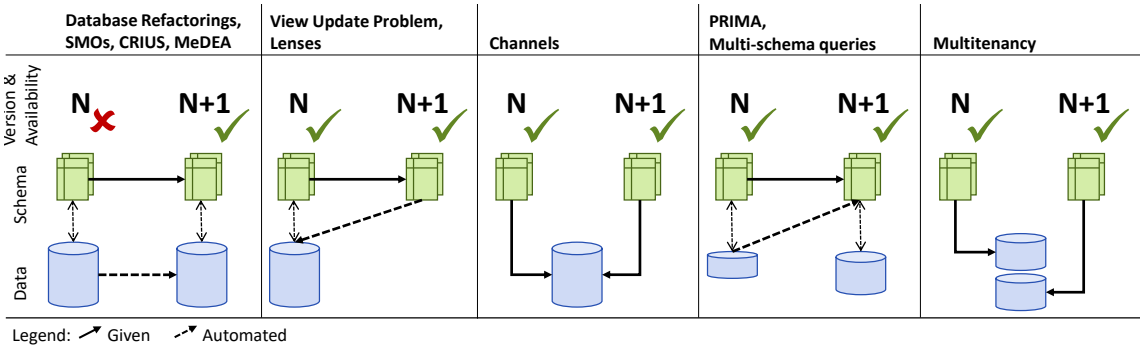


Figure 3.2: Related work regarding database evolution support.

and primary key and foreign key constraints with the same charm and simplicity as SMOs [41, 37]. Further, PRISM++ facilitates update rewriting, which allows to automatically adapt applications working on an old schema version to correctly access the data of a new SMO-evolved schema version. PRISM and PRISM++ are by design restricted to the plain forward evolution, however, they can serve as a solid starting point for more advanced SMO-based DELs. Another interesting research project originating from PRISM targets at the performance optimization of the actual evolution process [91]: the result is that column-oriented databases are—performance vice—perfectly suited for database evolution, since most evolution scenarios merely add or remove columns. This motivates to integrate the database evolution support deeper in the system and to utilize physical optimizations.

Since, both PRISM and PRISM++ merely focus on the forward evolution of the database, the data is physically always stored in the newest version which is also the only accessible schema version. An extension of this approach, called **PRIMA** [90], also allows querying historical data, which can be seen as a first step towards co-existing schema versions based on an SMO-based DEL. In order to increase the archive quality of a database, information from old schema versions is not physically transformed to a new schema version, as this may entail information loss for non-information preserving SMOs, but the data remains in the schema version where it has been created. Developers can now write temporal queries according to the newest schema version and PRIMA automatically translates them unidirectionally backwards to the former schema versions and incorporates their results. PRIMA does not support co-existing schema versions with bidirectional data propagation and enforces one fixed materialization. Nevertheless, the concept of decoupling the physical schema from the query execution on the logical schemas is very appealing.

In comparison to manually evolving a database with standard SQL, an SMO-based DEL is way easier to learn and to use. However, it still requires a technical understanding of the database evolution process. Therefore, the **CRIUS** project provides simple database evolution where non-technical users are equipped with an intuitive graphical user interface to easily extend a running database [94]. The database evolution is restricted to adding tables and columns which at least allows to grow the database with the application and prevents from common mistakes. Another interesting compromise between expressiveness and user-friendliness is model-driven database evolution with **MeDEA** [45]. MeDEA equips developers with an editor to specify schema changes in an Entity-Relationship diagram and the respective changes are automatically mapped to the underlying relational database model.

Co-existing versions: Traditionally, developers would create new virtual schema versions by merely writing views that represent the new schema version manually. Using standard SQL seems to be the natural way to describe the new schema version depending on the old schema version. The major challenge with this approach is the **View Update Problem**. While the views of the newly created schema version are obviously readable, they are not necessarily writable: Assume a view as a horizontal partition of the `Task` table that only contains tasks with the highest priority 1 similar to our

running example from Figure 2.4 on page 28. This selection predicate is not necessarily a constraint on the new partition, so users can add a new task with the priority 2 to this partition view. Translating this view update to the base table seems to be trivial as the new task can be simply written back to the base table `Task`. However, when reading the view again, the newly inserted tuple will be lost, since it does not satisfy the selection predicate of the view any longer. To resolve such issues, developers find themselves manually writing delta code, e.g. instead-of triggers and auxiliary tables and more fine-tuned view definitions, to explicitly make views writable. Hence, it is not surprising that simplifying and automating the view update problem is a long studied topic in the database research community.

Back in 1978, Dayal defined specific criteria which need to be satisfied by an updatable view [43]. Most importantly, the update must be performed correctly and there must be no side effects. Assume a view is defined by $v(D)$ on the base tables D, and Δ is an update issued on the view, and Δ^t is the transformed update that is physically executed on the base tables. The view on the updated base tables should be identical to virtually applying the update Δ directly on the view:

$$\Delta(v(D)) = v(\Delta^t(D)) \tag{3.1}$$

Thus, the view update problem is to find the correct Δ^t for a given Δ. From a user's perspective, this means that an updatable view behaves exactly like a standard table in a standard database. So any data inserted to a view must occur in the view; any data updated in a view must remain updated, and any data deleted from a view must not be visible when reading the same view again. Moreover, the intended update of the user's write operation must be the only change visible in the view—there must be no side effects like additionally occurring tuples and so on.

In 1981, Bancilhon et al. introduced the notion of a **view's complement** as "the information not visible within the view" [15]. For side-effect-free updates, this complement must not be affected by any write operation on the view. This can be conceptually validated by defining a second view that contains all information from the source schema minus the information in the first view. This auxiliary view must not be changed when writing to the actual view and propagating the changes to the actual table. Three years later, Cosmadakis et al. studied algorithms to compute a correct and side-effect-free update propagation and showed that the calculation of the view's complement is an NP-complete problem [36].

Since then, the view update problem is under constant investigation, e.g. Keller et al. proposed an **interactive view definition** mechanism, which polls developers when writing a view for the intended update propagation strategy [74] and more recently, Franconi et al. defined formal criteria for the updatability of given view definitions as well as an algorithm for the efficient computation of view updates [52]. In practice, common DBMSes make certain views automatically updatable as long as they satisfy a set of conditions. The respective documentations are accordingly long and complex: E.g. MySQL[4] requires a one-to-one relationship between the view's and the table's tuples, so there is a long list of restrictions ranging from the obvious exclusion of aggregation to complex restriction on joins and subqueries. In sum, after 40 years of intensive research, an easy-to-understand and powerful solution for the view update problem is still missing.

We will not review the whole body of research literature regarding the view update problem here, but point the interested reader to the book "View Updating and Relational Theory – Solving the View Update Problem" from Date [42]. Date summarized the research on the view update problem in his pointed remark "There is no magic". It is inherently hard to retrieve the correct and side-effect-free translation of updates from views to the base tables automatically, when the view is specified with relational algebra or SQL expressions. The initial intention of the view definition is lost between the lines and view definitions do not contain information on how to propagate data backwards from the

[4] `https://dev.mysql.com/doc/refman/5.7/en/view-updatability.html` (Accessed on 31.08.2017)

view to the tables. The bidirectional SMOs used to describe the evolution in multi-schema-version database systems change these fundamental restrictions by design and allow to create updatable views immediately. Developers merely have to accept the overhead of learning an SMO-based DEL and of defining the view as a bidirectional evolution from the source tables.

An advanced formal approach to define and solve the view update problem are lenses, which are a well recognizes theoretical framework using the intuition of an optical lens between a source and a target data structure that can be transformed into each other [50]. Particularly, **Relational Lenses** assume a lens between the view and its base tables, which allows to describe and solve the view update problem and further facilitates co-existing schema versions [26]. Relational lenses guarantee that data written in the view survives the round trip over the base tables unchanged. The relational lenses can be composed of lenses that correspond to the relational algebra operations selection, projection, and join but are bidirectional. Thereby, they carry an explicit update policy for the backward propagation from the view to the base table and solve the view update problem—at least for those evolutions that can be expressed with the given set of lenses. The limitations of using relational lenses for multi-schema-version data management are their restricted expressiveness and the fixed materialization of the data at the source side of an evolution.

Similar to the discussed relational lenses, further approaches for co-existing schema versions within the same database have been proposed in the literature. For instance, Terwilliger et al. introduced the concept of **Channels** that allow creating multiple virtual databases all based on one physical database underneath [111, 107]. The authors define similar bidirectionality conditions as the discussed relational lenses—however, on different meta levels, particularly (1) between the database and the application: data written in the application and persisted in the database should be identical when reading it again in the application and vice versa; and (2) between the virtual database schema and the physical database schema: data written on any side needs to survive one roundtrip completely and correctly. The central idea of channels is to have virtual databases that behave like common relational databases from the application's perspective at run time. At development time, the developers use so called Virtual Databases Channels, similar to our SMOs, to stepwise map the envisioned virtual database to the one fixed physical database schema. Those channels can merge and partition a table both vertically and horizontally as well as pivoting and unpivoting a table as well as calculating columns based on an invertible function. This allows the virtual databases to process any read/write access as well as simple schema modifications in the virtual databases as if they were real physical ones. Under the hood, all accesses are transformed through the channels to the physical database schema, which lays special focus on the efficient implementation of the transformation of read/write accesses. This has been extensively studied in [20]. In contrast to multi-schema-version database systems, channels require one fixed physical database schema by design; changing it would require to redefine all channels defined so far. Multi-schema-version database systems enable full data independence which allows co-existing schema versions independently from the actual physical database schema.

The wish for co-existing schema versions occurs in different areas with different flavors: Interesting research areas targeting at co-existing schema versions as e.g. multi-schema querying support and multitenancy databases. Let us shortly discuss these two representatives. To persist and query multiple schema versions within the same database, Grandi published **Multi-schema Querying** already back in 2002: He proposes both a data model and an SQL extension to support multiple schema versions within one database [55]. The data model relies on one physical schema for each logical schema version, but data objects occurring in multiple versions are not necessarily replicated. So, for querying this data, an SQL extension for querying multiple schema versions adds a schema version identifier for each table in the SQL query. When accessing a specific table in a specific schema version, the data is collected from all other related physical schema versions as well by propagating it along the schema version history.

In **Multitenancy Databases**, the data of multiple customers of a cloud application is stored within the same database. Based on the observation that the data of thousands of similar applications can be managed best within one database schema instead of thousands of database schemas, the challenge is to consolidate those thousands of similar but slightly different schemas into one. In 2008, Aulbach et al. proposed a set of generic schema designs ranging from base tables that all tenants share with extensions for groups of tenants, over pivot tables and universal tables, all the way to chunk tables that split up the individual schemas into common chunks [11]. A major criteria for choosing the right schema design for a given set of tenants, is the performance for the individual tenants, which has been extensively studied and evaluated in [12]. In multitenancy databases, multiple applications can access different schema versions, however, each tenant sees only the individual subset of the data and the physical table schema is fixed at development time, which clearly distinguishes them from the envisioned multi-schema-version database systems. In 2011, the authors took the next step and allowed also data sharing between all tenants [13]. To this end, a **Polymorphic Table** is introduced that can grow in two directions: First, there is a shared common schema, but each tenant can add individual columns to the polymorphic table. Second, there is a shared set of tuples that all tenants see and each tenant can add individual tuples to the polymorphic table. The individual columns and tuples are exclusively visible for the respective tenant while the shared columns/tuples are visible for all tenants. Polymorphic tables enable co-existing schema versions on the same data set, however, the physical table schema is still fixed and the expressiveness of the evolution of a tenant's schema is basically restricted to adding columns.

NoSQL stores use more flexible schemas to easily persist heterogeneous and changing data [84]. One anticipated benefit was to make schema evolution obsolete and thereby greatly support agile software development, since no big upfront modeling of the database schema is required anymore. Nevertheless, all data stores—even NoSQL stores—have a schema that may be hidden in the data itself or in the application and this schema will evolve eventually leaving developers with the same problems as relational database evolution but with even less tool support. Scherzinger et al. consider the **schema evolution in NoSQL stores** that have identifiable entities with attribute-value pairs similar to the Google Datastore [69] and propose a general evolution language for such stores [101]. Special emphasis is laid on the actual migration of the data after a schema evolution. It is not always necessary to migrate an entity eagerly to a new schema version, but the entities can be migrated lazily as well [102]. As each entity is individually identifiable, the lazy data migration allows migrating individual entities just when they are accessed. Scherzinger et al. translate the schema evolution into sets of Datalog rules; when accessing an entity in a new schema version, it is propagated through these Datalog rules to the current schema version in a very safe and robust manner.

Document stores for e.g. XML or JSON data are another commonly used flexible data storage. Klettke et al. developed a conceptual model for the XML schema evolution providing a graphical editor for the developers to intuitively specify the schema evolution. The instances, so all XML documents, are then automatically evolved to match the new schema version [76]. Further, Geneves et al. provide a framework that takes an XML schema evolution and analyzes its actual effect on the XML documents as well as on queries that were written against the old schema version [53]. To have co-existing schema versions in an XML store, we merely need to apply the concept of relational updatable views to XML views [85]. A comprehensive survey regarding the view update problem in XML views was published by Chen in 2011 [30].

The problem of database evolution for **Data Warehouses** is studied almost as much as for relational databases in general. The special data model of data warehouses with facts, dimensions, and hierarchies etc. calls for specialized database evolution support. There are at least 15 different evolution languages for data warehouses that all support different aspects; they are all carefully compared and presented in a survey published by Arora et al. in 2011 [10]. For instance Kaas et al. propose a language that allows adding and deleting both dimensions and levels and measures as well as connecting and disconnecting attributes to/from dimension levels [72]. A step towards co-existing schema versions is taken e.g. by Golfarelli who supports queries spanning multiple schema versions within a data warehouse [54]. The underlying evolution language allows adding and deleting attributes as well as functional dependencies.

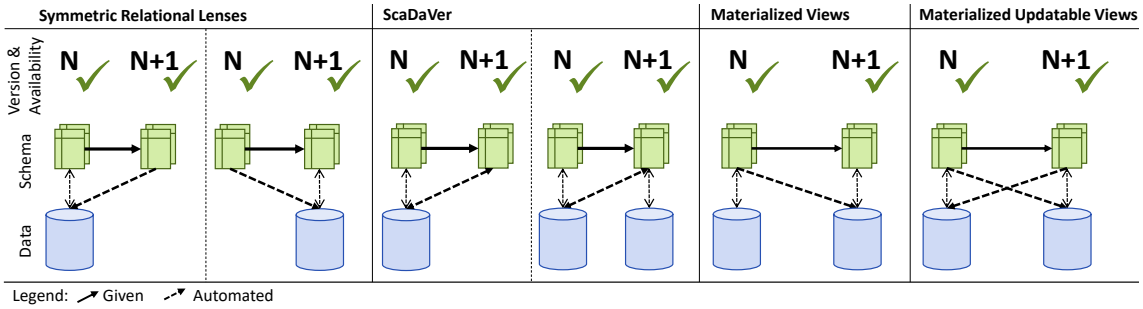


Figure 3.3: Related work regarding data independence for multi-schema-version database systems.

3.3.2 Full Data Independence for Multi-Schema-Version Database Systems

After discussing related work on creating and accessing multiple co-existing schema versions within a single database, we will now focus on the materialization of the data that is shared among multiple schema versions. The solution space ranges from fully redundant materialization of all schema versions to non-redundant materialization in a freely configurable subset of the table versions. Exploiting this solution space and freely changing the physical materialization without limiting the availability of any co-existing schema version is **Data Independence** lifted to the requirements of multi-schema-version database systems.

Symmetric Relational Lenses as introduced by Hofmann et al. take a step towards data independence on the solid formal ground of relational lenses [65]. Symmetric relational lenses maintain a complement, as introduced for the view update problem in the previous section, on both sides of the lens. Hence there is no explicit data table and view side any longer, but both sides can store the exact same information—with different structures—and can be mapped to each other. The only difference between the two sides of the symmetric relational lens is that evolution-wise there exists an old (the source) and a new (the target) schema version. There are two conditions that need to be satisfied for any symmetric relational lens. Assume $\gamma_{trg}(D, C)$ maps source data D and the complement C to the target-side data D' and an updated complement C'. Intuitively speaking, we store data from the source side at the target side which requires the complement, as lenses are not necessarily information preserving. When reading back this data at the source side with $\gamma_{src}(D', C')$, the lens guarantees that we can completely and correctly read the data D as it was in the first place—the data survives one round trip. Further, the complement C' reaches a fix point when propagating the data back to its origin, so we do neither lose nor gain any data during a round trip. The same conditions needs to hold vice versa.

$$\frac{\gamma_{trg}(D, C) = (D', C')}{\gamma_{src}(D', C') = (D, C')} \tag{3.2}$$

$$\frac{\gamma_{src}(D, C) = (D', C')}{\gamma_{trg}(D', C') = (D, C')} \tag{3.3}$$

Hence, these two conditions ensure that, no matter on which side of a lens we actually store the data, both sides behave like regular database tables and write operations are side-effect free and persisted correctly. In this thesis, we adapt these two conditions to formally define and guarantee the data independence of a bidirectional SMO-based DEL. Symmetric relational lenses are a sound formal framework for robust co-existing schema versions; however, relational lenses are very limited in their expressiveness and do not capture the developers' intentions like SMOs do. Hence, they are not very appealing for multi-schema-version database systems from a practical point of view.

The **ScaDaVer** [116, 115] system allows comfortable version management for database schemas similar to version control systems like SVN and GIT. Particularly, ScaDaVer allows to evolve the database schema by adding, renaming and removing both columns and tables. The core schema version can be evolved to branches as well, which can be merged back into the core later on. This bridges the gap in the continuous version management of database development and the actual application development. Any data from an old schema version will be propagated forwards to the newer schema versions, which raises the question, how to store the data physically: redundantly in both the new and the old version or non-redundantly only in the old version. The author of the ScaDaVer system extensively evaluates these two possibilities [115].

Storing the data redundantly in multiple schema versions is tightly coupled to the problem of **Materialized View Maintenance**. We already discussed techniques to solve the view update problem; so within the boundaries of updatable views, we can create co-existing schema versions merely by defining new schema versions as updatable views on the initial schema version. Assume we store the data redundantly by materializing the view. Then, updates to the base table can become very expensive when using the naïve approach of recalculating the whole view from scratch. There is an active research community aiming at making this update incremental and more efficient. Specifically, an update to the base tables should be translated into a minimal update on the materialized views without recalculating the whole view. For instance Gupta et al. gave a detailed overview of the literature on materialized view maintenance [56]. A recent advance is **DBToaster** [77], especially focusing on very frequent changes to the base tables. It statically analyses the view definitions using recursive finite differencing, to obtain small and incremental update rules for the materialized views. This technique allows the authors to refresh tens of thousands of views within a second.

Idris et al. propose an algorithm to update materialized views in linear time—at least for certain classes of conjunctive query views [68]. The presented algorithm is based on the well-established Yannakakis Algorithm [121], which e.g. minimizes joins and allows enumeration in constant time for many queries. The authors present a **dynamic version of the Yannakakis Algorithm**, which can be used to maintain materialized views in linear time w.r.t. the size of the base tables. Another more general approach has been published by Behrend et al., which uses the formal notion of Datalog rules for the view definition and thereby facilitates simple and efficient calculation of incremental updates to a recursively defined view based on Datalog [19]. When updates occur on the base tables—the facts of the Datalog rules—these changes are propagated through the Datalog rule using Magic Sets. The computation of propagating updates through the Datalog rules from the facts to the final view is simplified on different levels of granularity and accuracy, which can be used to simplify the repeated evaluation of complex subqueries and thereby significantly reduces the number of tuples produced by the Datalog rules during the computation. All discussed approaches for (incremental) materialized view maintenance are important for multi-schema-version data management, since the ideas can be applied for the incremental propagation of updates between schema versions as well. If we combine the discussed techniques with updatable views, we already have co-existing schema versions with a flexible materialization underneath—however, we are still forced to materialize the base tables, so the oldest version will always be materialized. Further, the expressiveness of the view definitions is limited to languages for updatable views. Hence, materialized updatable views are powerful but still clearly differ from true multi-schema-version database systems such as InVerDa.

Materializing a view can significantly speed up read operations The downside is that updating materialized views along with their base tables implies an overhead for writing. Thus, there is a trade-off between read performance, write performance, and space constraints. The optimal choice of materialized/non-materialized views greatly depends on the current workload but is crucial to obtain a reasonable overall performance. There is a multitude of tools and publications regarding **Materialized View Selection** presenting a wide range of different approaches. As examples, Mistry et al. realize materialized view selection based on multi-query optimization techniques [89], and Agrawal

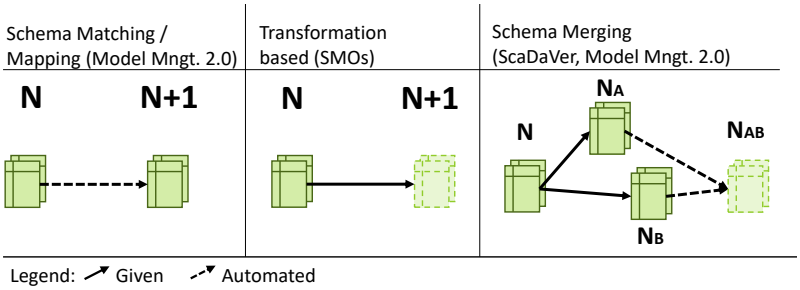


Figure 3.4: Related work regarding co-evolution and schema versions management.

et al. tuned a cost model-based approach to be robust and to scale [4], so it was incorporated into Microsoft's SQL server 2000 [5]. An important objective for multi-schema-version database systems is an adviser to find the best materialization for a given workload—a problem that is very similar to the materialized view selection problem and can also be solved cost model-based. The difference, however, is that there are no base tables in multi-schema-version database systems that are always forced to be materialized like the base table of potentially materialized views.

3.3.3 Co-Evolution

For the proper evolution management of relational database schemas, Bernstein et al. introduced **Model Management 2.0**: concepts and tooling for handling and processing multiple schema versions during development. This includes a large variety of operation on schema versions like DIFF, MERGE, and INVERT [25]. With InVerDa, PRISM, ScaDaVer, etc. most systems automatically generate new schema versions based on the given transformations—e.g. sequences of SMOs—in a "before the fact" manner. In contrast, Model Management 2.0 allows deriving the actual evolution "after the fact" from given schema versions [6]. Besides calculating the difference between two given schema version, Model Management 2.0 also allows merging two schema versions by calculating the mappings from the last common schema version and combining these mappings using mapping composition techniques [23, 46]. A major objective for multi-schema-version database systems is that they also allow merging two schema versions, however, we do not want them to guess the intended evolutions but let the developers explicitly specify it with the help of SMOs; this allows to maintain the intended evolution way better in the merged schema version. Also, calculating the inverse of a derived mapping to propagate data backwards through multiple evolution steps is a hard problem due to the ambiguity of inverses and compositions of mappings [47]. Ideally, we would need to evaluate any possible inverse or composition of mappings against valid data and queries, which is often not feasible in practice, so the inverse and composition of mappings can only be estimated as close as possible; with SMO-based evolution we avoid all these problems by design as the inverse and composition of SMOs is well-defined in the SMOs' semantics and obvious to the developers.

To obtain the discussed mapping between two given schema versions, there is a huge research field about **Database Schema Matching** [96]. In 2001, three outstanding researchers in the field of schema matching Jayant Madhavan, Phil Bernstein, and Erhard Rahm, published a schema matching system called Cubid [86]. Cubid relies on linguistic analysis of the two given schema versions to match columns and tables as well as graph matching algorithm to match the structure of the two schema versions. Ten years later, the same three authors looked back how the field of schema matching evolved [24]. Their observation was that it is a constantly hot research topic, since there are always new source of information available to increase the quality of the schema matching. Particularly, they state that "the problem of schema matching is inherently open-ended", which assured us in following the SMO-based approach where developers explicitly provide the mapping between two schema

versions as a sequence of SMOs and no schema matching techniques are required. According to surveys, available schema matching techniques cover linguistic matching, instance based matching, structure-based matching, rule-based matching, matching based on similarity measures, and deep-learning approaches as well as all kinds of combined hybrid techniques [103, 96]. So, we can safely say that schema matching is an important and active research field, but staying clear of rocks by specifying the evolution between schema versions explicitly with an SMO-based DEL facilitates a more stable and robust solution.

3.3.4 Summary

As can be seen, there is a wide range of different approaches for database evolution support. Figure 3.2 summaries the discussed approaches for proper database evolution and co-existing schema versions. We started out with database refactorings, SMOs, and model-based evolutions as pragmatic means to describe the forward evolution of a database schema including already existing data. These techniques allow a robust forward evolution of the database, however, the old schema versions are no longer accessible after the evolution has been executed. With updatable views, which can be formally described using e.g. relational lenses, we take a first solid step towards co-existing schema versions, as both the base tables and the views are fully accessible for applications. Hereby, the view is the evolved schema version.

Afterwards, we discussed further techniques for co-existing schema versions that all rely on different physical materializations of the data—however, none of them offers a flexible materialization so far. While channels statically map different logical schema versions to one fixed physical schema, multi-schema query systems, such as PRIMA, maintain the data non-redundantly in those schema versions where it was written and merely rewrite queries to collect data from all schema version in the system. Another discussed approach are multitenancy databases, where multiple schema versions are represented within a more flexible schema; the flexibility is achieved by e.g. having unnamed or even untyped columns ready to be used by every schema version individually, so the actual schema is captured at the application layer.

All the approaches presented in Figure 3.2 (partly) support co-existing schema versions, while their physical schema is always fixed. Therefore, we extended the discussion to related work regarding full data independence for co-existing schema versions, as summarized in Figure 3.3. Namely, symmetric relational lenses extend the already discussed relational lenses to have two equally important sides—there is no distinction between base tables and views any more, but data can be stored at any side and accessed on both sides of the lens. The main drawback of symmetric relational lenses is their limited and rather unintuitive expressiveness, which makes them hard to establish in multi-schema-version database systems. Another step towards data independence for co-existing schema versions has been taken with ScaDaVer: Having multiple co-existing schema versions, ScaDaVer allows the developers to decide whether data should be replicated or propagated between these versions. Furthermore, materialized views also provide an additional degree of freedom in the materialization of co-existing schema versions given the views are made updatable as discussed before.

Finally, Figure 3.4 summarizes the presented related work regarding the management of different schema versions within a database system. We discussed schema mapping and schema matching techniques that are used to extract the evolution between two given schema versions. A famous representative is Model Management 2.0, which allows not only to extract the mapping between given schema versions but also to merge different schema versions. As an alternative approach to extracting the schema mapping after the fact from given schema versions, we also surveyed transformation based

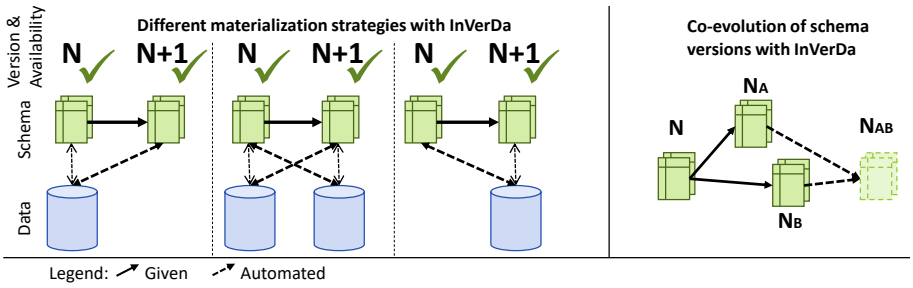


Figure 3.5: InVerDa's features as comparison to related work.

approaches: When developers use SMO-like descriptions in the first place, the transformation is explicitly given and can be used further for e.g. merging two schema versions without the fuzziness of automatically extracted mappings. ScaDaVer, for instance, allows merging schema versions based on the explicitly given transformations—however, only for a very limited set of SMOs. Summing up, the wide range of discussed related work covers the main objectives for multi-schema-version database systems individually, but a solution that unites the benefits of all objectives is still missing and not trivial as we will show in the next section.

3.4 EVALUATION OF OBJECTIVES

As a summary of the discussed related work, we collect the respective evaluation of the objectives in Table 3.1. There are many practical and theoretical results towards multi-schema-version database systems, however, to the best of our knowledge and as can be seen in the table, there is currently no solution that covers all objectives. In the following, we go through the single objectives to summarize the state of the art and highlight the distinction of InVerDa. For comparison, Figure 3.5 pictures InVerDa's capabilities for multi-schema-version data management.

Database evolution support: As motivated before, the envisioned database evolution support in multi-schema-version database systems should be based on a **DEL** that explicitly captures the transformation/evolution from one schema version to another. In the surveyed literature, there are many promising approaches like database refactorings and SMO-based DELs, however, none of the existing languages is both **practically complete** and **relationally complete**. Those approaches that are both practically and relationally complete, in contrast, do not intuitively couple the evolution of both the schema and the data, which makes e.g. data propagation and merging of schema versions unnecessarily ambiguous. InVerDa's BiDEL is SMO-based and has proven to work for practical scenarios and it is formally guaranteed to be relationally complete, which justifies the claim that there will be no need to fall back on traditional manual database evolution and the benefits of multi-schema-version database systems can be used for each and every upcoming scenario. This is an essential requirement to make multi-schema-version data management practically applicable.

Further the support for **co-existing schema versions** is still very limited in the literature. While most of the presented solutions support multiple readable schema versions in one database, write operations are usually only propagated forwards. So, the existing data is able to follow the evolution in forward direction, but any data written in newer versions is not visible in older versions. The only exceptions are updatable views and symmetric relational lenses. Since these approaches have a very limited expressiveness, so many scenarios cannot be expressed at all, they are not practically applicable in multi-schema-version database systems, yet. Nevertheless, the research about the view update problem and symmetric relational lenses defined fundamental requirements for concurrently

updated schema versions with bidirectional propagation. In a nutshell: each schema version itself has to act like a full-fledged single-schema database. We formally validate these conditions to hold for InVerDa, so we guarantee all the established transaction and persistence guarantees known from relational databases also for multi-schema-version database systems based on the powerful SMO-based evolution language BiDEL.

Full data independence for multi-schema-version database systems: The data independence in multi-schema-version database systems is not explored in much detail so far, which is not surprising after all, as the precondition of co-existing schema versions is not even fully explored. Namely, symmetric relational lenses provide a solid formal framework for a **flexible materialization**, so data can be stored either at the source side, or at the target side, or even at both sides of an evolution. With InVerDa, we apply the power of this flexible materialization to the simplicity and expressiveness of the SMO-based BiDEL. This enables DBAs to freely move the physical materialization along the schema version history without affecting the logical schema versions, so InVerDa provides full data independence in multi-schema-version database systems—we are not aware of any other work with this contribution.

✓ supported
● partly supported
✗ not supported

	Database Evolution Language (DEL)	*Practical completeness*	*Relational completeness*	*Co-existing versions (forwards)*	*Co-existing versions (bidirectional)*	*Flexible materialization*	*Adviser for materialization*	*Guaranteed correctness*	*Co-evolution*	*(Semi-)automatic co-evolution*
Agile/Refactorings	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
SVN/GIT	✗	✗	✗	●	✗	✗	✗	✗	✓	●
Software product lines (SPL)	●	✗	✗	●	✗	✗	✗	✗	✗	✗
Variable data model for SPLs	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
DaVe	●	✗	✓	✓	✗	✗	✗	✗	●	✗
SQL	✗	✓	✓	●	●	✗	✗	✗	✗	✗
Database refactorings	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
PRISM/PRISM++	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
CRIUS/MeDEA	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Updatable views	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗
Materialized views	✗	✓	✓	✓	✗	✗	✗	✓	✓	✗
Materialized updatable views	✗	✓	✓	✓	✓	✗	✗	✓	✓	✗
Materialized view selection	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗
Multi-schema queries/PRIMA	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
Relational lenses	●	✗	●	✓	✗	✗	✗	✓	✗	✗
Symmetric relational lenses	●	✗	●	✓	✓	✓	✗	✓	✗	✗
Multitenancy	✗	✗	✗	✓	✗	●	✗	✗	✗	✗
ScaDaVer	✓	✗	✗	✓	✗	●	✗	✗	✓	●
Model Management 2.0	✗	✗	✓	✓	✗	✗	✗	✗	✓	●
InVerDa	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Contribution and distinction from related work.

Having the freedom of a flexible materialization calls for an **adviser** that automatically determines the best materialization for a given workload. This is a common problem in materialized view selection; when we consider the views as evolved table versions, the question of finding the best set of materialized table versions basically boils down to materialized view selection. The problem differs as materialized view selection always needs to materialize at least the base tables, which does not hold in a multi-schema-version database system. Thus, InVerDa adopts ideas, like the cost model-based optimization, but additionally considers the special characteristics of bidirectionally and data independently propagating data access operations through SMOs.

Another objective resulting from co-existing schema versions with a flexible materialization is the **guaranteed correctness** of data accesses to any schema version. This condition is met by updatable views in any relational database that ensures transaction guarantees for such views. Further, symmetric relational lenses again lay a solid formal foundation to validate the correctness for a restricted class of view definitions. So, all the solutions discussed in the literature provide correctness for co-existing schema versions. Their limitations w.r.t. multi-schema-version database systems are both a restricted DEL and an inflexible materialization. InVerDa brings these worlds together and formally guarantees that each single schema versions behaves correctly like a common database.

Schema version management: Finally, the objective of **co-evolving schema version** and to do so **(semi-)automatically** based on the developers' intentions are tough challenges in multi-schema-version database systems. Code version control systems like SVN and GIT allow to consolidate the evolution of a local copy and the evolution of the global copy, which basically merges the two versions into a new one. With ScaDaVer and especially Model Management 2.0 there are also very active research projects around the co-evolution of database schemas; however, these approaches are either very limited in the expressiveness of the evolution or require a lot of manual effort. To the best of our knowledge, InVerDa' miX is the first approach to automate this process based on the developers' intentions captured in BiDEL's SMOs.

Summing up, the active research community targeting evolution in software and database development came up with significant results for all the individual objectives of multi-schema-version database systems. To the best of our knowledge, there is no solution that combines all the desired benefits. InVerDa does so based on the SMO-based evolution language BiDEL. Using the rich semantics of BiDEL's SMOs, InVerDa can automatically generate all the delta code required to provide co-existing schema versions with a flexible materialization and formal guarantees like the relational completeness of BiDEL and the correctness of each single co-existing schema version. Further, merging schema versions based on their intentions is a unique feature of InVerDa that relies on the rich semantics of BiDEL's SMOs as well. This multitude of powerful features in a system that is actually implemented and practically applicable is the unique contribution of this thesis.

4

EVOLUTION OF SCHEMA VERSIONS WITH BIDEL

Multi-schema-version database systems, such as InVerDa, allow handling multiple schema versions within the same database. Developers create those schema versions either by creating new and empty tables from scratch or by evolving an existing schema version to a new schema version using a DEL or by merging two existing schema versions into a new one. The latter option of merging schema versions will be discussed in Chapter 5. For the creation of new schema versions, we now introduce a DEL that allows both creating new tables and evolving existing tables and even propagating data between these table versions in order to let multiple schema versions co-exist. We will discuss the objectives of such a DEL in Section 4.1, present the syntax and semantics of our language BiDEL in Section 4.2, evaluate the initial objectives in Section 4.3, and finally summarize the chapter in Section 4.4.

4.1 OBJECTIVES

A comprehensive and powerful DEL can greatly simplify a database developer's life since it serves as an intuitive documentation, prevents mistakes, and facilitates to automate further tasks that come along with the database evolution. We identified three major characteristics a DEL needs to satisfy to be suitable for multi-schema-version database systems: a DEL should (1) couple schema and data evolution to explicitly represent the intention of the database evolution, (2) be complete to prevent developers from falling back on traditional SQL, and (3) be bidirectional to facilitate co-existing schema versions. SMO-based DELs couple the data evolution to the schema evolution by design—thus, they implicitly satisfy the first characteristic. The completeness of a DEL can be evaluated from multiple aspects, particularly we cover the intuitive practical completeness as well as the formal relational completeness in more detail (Section 4.1.1). Finally, the bidirectionality of a DEL facilitates the propagation of data both forwards and backwards which is an essential prerequisite for co-existing schema versions that all represent the same conceptual data set (Section 4.1.2). To be bidirectional, some SMOs provide additional parameters, which the developers only need to specify if the bidirectional propagation of data accesses—hence truly co-existing schema versions—are required.

For the general **taxonomy** we use within this work, a DEL $\mathcal{L}$ is a set of SMOs with parameters to be instantiated. For instance, the SMO to drop a column from a table requires two parameters: the name of the table and the name of the column; if bidirectionality is required, we need an additional parameter: a function calculating the values for the dropped column in the source version when tuples are inserted in the target version without the respective column. Let $inst\,(\mathcal{L})$ be the set of all operation instances of $\mathcal{L}$ with valid parameters. Then, a relational database $D = \{R_1, \ldots, R_n\}$ with tables R_i can be evolved to another relational database $D' = \{R'_1, \ldots, R'_m\}$ with the forward mapping function γ^s_{trg} of an SMO $s \in inst\,(\mathcal{L})$, which is denoted as $D' = \gamma^s_{trg}(D)$. Given a sequence of SMOs $S \in inst\,(\mathcal{L})^+$ with $S = (s_1, s_2, ...s_n)$ a given database D is evolved to another database $D' = \gamma^{s_1}_{trg}(\gamma^{s_2}_{trg}(...\gamma^{s_n}_{trg}(D)))$, which we formally denote as $D \xrightarrow{S} D'$. For bidirectional SMOs the mapping function γ_{src} denotes the backward propagation of data analogously to γ_{trg}.

4.1.1 Completeness

A complete DEL allows the developers to intuitively specify any intended evolution with the given set of SMOs. Whenever the developers have to fall back on traditional SQL, all the formerly solved problems return, making completeness an essential requirement for the practical applicability of InVerDa. We consider two levels of completeness: practical completeness and relational completeness.

Practical Completeness Practical completeness is a general and intuitive measure indicating the feasibility of a DEL for common database evolution scenarios. Given the evolution history of existing projects, a DEL must allow to model the same evolution exclusively with the given set of SMOs. As an example, there is a detailed benchmark provided by Carlo Curino et al. [40] using the evolution history of 171 versions of Wikimedia—the backend of e.g. Wikipedia. Any practical complete DEL should allow to model the evolution of both the schema and the data completely and correctly as originally intended by the developers. Practical completeness evaluates the completeness of a DEL after the fact with respect to known evolutions; there is no guarantee that prevents developers from hitting limitations of the DEL for uncommon evolution scenarios.

Relational Completeness Relational completeness is a formal property. It guarantees that any evolution that can be expressed with relational algebra expressions can be expressed with the DEL's SMOs as well. We introduce a minimal relational complete DEL, called $\mathcal{L}_{\text{min}}$. This allows to evaluate the relational completeness of a DEL later on by showing that it is at least as powerful as $\mathcal{L}_{\text{min}}$.

The relational completeness of a DEL is a fundamental prerequisite, as the advantages of DELs can only be used when the developers specify all evolution steps with the given set of SMOs. Therefore, we have to make sure that there is no need for developers to fall back on traditional SQL in any situation. We take the relational algebra as the key reference for the expressiveness of traditional SQL. A minimal language providing relational completeness is $\mathcal{L}_{\text{min}} = \{\texttt{Add}(\cdot,\cdot), \texttt{Del}(\cdot)\}$ with

$$\begin{aligned}\texttt{Add}(R', \epsilon) &\rightarrow D \cup \{R' = \epsilon(R_1, \ldots, R_n)\}\\ \texttt{Del}(R) &\rightarrow D \setminus \{R\}\end{aligned}$$

The $\texttt{Add}(\cdot,\cdot)$ operation adds a new table R' to the database D based on the given relational algebra expression ϵ that works on the relations of D. The $\texttt{Del}(\cdot)$ operation removes the specified table R from D. A database D can be evolved to any other database D' with a sequence $S \in inst(\mathcal{L}_{\text{min}})^+$, where the tables in D' are computed from D with relational algebra expressions ϵ in the $\texttt{Add}(\cdot,\cdot)$ operation. Thus, $\mathcal{L}_{\text{min}}$ is relationally complete. From a practical standpoint however, $\mathcal{L}_{\text{min}}$ is not very appealing, because it is rather unintuitive and not oriented on actual evolution steps. However, any other DEL that is as expressive as $\mathcal{L}_{\text{min}}$ is relationally complete as well.

To the best of our knowledge, one of the most advanced DEL designs is PRISM/PRISM++ [39, 41]. PRISM++ provides SMOs to create, rename, and drop both tables and columns, to divide and combine tables both horizontally and vertically, and to copy tables. The PRISM++ authors claim practical completeness for their powerful DEL, by validating it against evolution histories of several open source projects. Although this evaluation suggests that PRISM++ is sufficient also for other software projects, it does not provide any reliable completeness guarantee—it is not shown to be relationally complete. For instance, we do not see an intuitive way to remove all rows from a table A, which also occur in a table B (relational difference) using the PRISM++ DEL, since it does not offer any direct or indirect outer join functionality. Thus, we consider PRISM++ not to be relationally complete.

4.1.2 Bidirectionality

The bidirectionality of the DEL is required to facilitate co-existing schema versions with data being propagated both forwards and backwards between them. While unidirectional DELs describe only the forward evolution, the SMOs of a bidirectional DEL also contain enough information to propagate data backwards. Let us take the view update problem as an example: the unidirectional view definitions precisely describe how to compute the view from the base tables. However, the backward direction is excluded; most views are not implicitly updatable since the initial view definition does not contain instructions on how to propagate data backwards. While the research community continuously proposes sophisticated techniques to (semi-)automatically estimate the intended backward propagation from the given view definition, data distribution, workload, structure, etc., bidirectional DELs explicitly require the developers to provide the backward propagation of data as well. Thereby, bidirectional DELs allow generating implicitly updatable views for new table versions, which is the basis for co-existing schema versions within a multi-schema-version database system.

4.2 SYNTAX AND SEMANTICS

As a practical complete and relational complete and bidirectional DEL, we propose BiDEL, which is the DEL used by InVerDa. Basically, BiDEL is an evolution language for relational databases to describe the evolution of both the schema and the data from a source schema version to a target schema version. As BiDEL also contains an SMO to create new tables from scratch, it can also create completely new schema versions without a source schema version. In its pure form, BiDEL allows to describe the **forward evolution** including the forward propagation of data with the expressiveness of the relational algebra.

However, BiDEL is not restricted to the plain forward evolution. Co-existing schema versions that work on the same conceptual data set, require **bidirectional semantics**—data needs to be propagated both forwards and backwards between all schema versions. To enable the backward propagation additionally to the forward propagation, BiDEL's SMOs may have additional arguments to provide the necessary bidirectional semantics. The developers need to provide those additional argument only if backward propagation is required. Namely the `DROP COLUMN` SMO, the `MERGE` SMO, and the `DECOMPOSE` SMO additionally require missing parameters of their inverse SMOs to propagate data backwards. These slight extensions are very limited and do not decrease the beauty and simplicity of BiDEL.

In this section, we define BiDEL's syntax including bidirectional SMOs—but we focus the discussion of the formal semantics on the forward evolution. For the semantics of the backward propagation, we apply the formal semantics of the inverse SMO and only intuitively describe the implications. The semantics of backward propagation will be formalized in Chapter 6 along with the propagation of read and write operations to the auxiliary tables.

Similar to the intuitive and field-proven design of PRISM++, BiDEL contains SMOs to describe coupled changes of both the schema and the data as units, which clearly distinguishes it from SQL-DDL and -DML. BiDEL evolves a maximum of two source table versions to not more than two target table versions, which keeps the language intuitive and easy to learn. BiDEL operations systematically cover all possible changes that can be applied to tables. To intuitively cover the whole space of possible evolution operations within a relational database, we first define the basic elements on which evolution operations are defined: Tables are the fundamental structuring element and the container for payload data in a relational database. Secondary database objects such as views, constraints, functions, stored procedures, indexes, etc. should be considered in database evolution as well. However, in this thesis we focus on the evolution of the primary data.

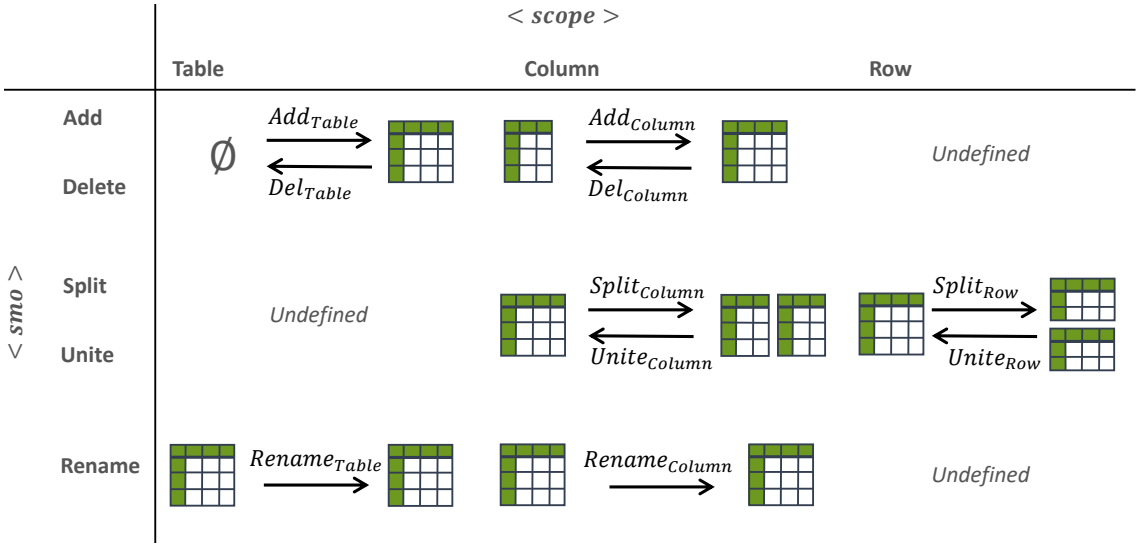


Figure 4.1: Structure of BiDEL's SMOs.

Our DEL BiDEL defines SMOs of the pattern $\langle smo \rangle_{\langle scope \rangle}(\Theta)$, where $\langle smo \rangle$ is the type of operation, $\langle scope \rangle$ is the general database object the operation works on, and Θ is the set of parameters the SMO requires. Figure 4.1 gives a systematic overview of all SMOs in BiDEL. A relational database table is a two-dimensional structure consisting of columns and rows, hence, SMOs can operate on the level of columns, of rows, or of whole tables. On all three levels there are five basic operations: `Add`, `Del`, `Split`, `Unite`, and `Rename`. We now introduce the meaningful operations, as shown in Figure 4.1. First, BiDEL has two basic operations to create ($\texttt{Add}_{table}$) and drop ($\texttt{Del}_{table}$) tables as a whole, similar to their counterparts in a standard SQL-DDL. Second, BiDEL has a set of operations to modify a table. BiDEL offers six table modification SMOs $\langle smo \rangle_{\langle scope \rangle}$ with $\langle scope \rangle \in \{column, row\}$ and $\langle smo \rangle \in \{\texttt{Add}, \texttt{Del}, \texttt{Split}, \texttt{Unite}\}$. For instance, $\texttt{Del}_{column}$ reduces the column set of a given table by removing one column and $\texttt{Split}_{row}$ partitions a table horizontally by splitting its set of rows, while $\texttt{Split}_{column}$ partitions a table vertically by splitting its set of columns. Creating and dropping rows are data manipulation operations that are already well supported by traditional SQL-DML and therefore out of scope for a DEL. BiDEL defines no `Split` or `Unite` of whole tables since these operations are restricted to either column or row scope. Third, BiDEL includes two SMOs to rename a table ($\texttt{Rename}_{table}$) and a column ($\texttt{Rename}_{column}$). The renaming of rows is undefined.

Regarding relational completeness, e.g. $\texttt{Rename}_{column}$, $\texttt{Rename}_{table}$, and $\texttt{Del}_{column}$ are not necessary, as they are subsumed by the remaining SMOs. However, they are very common [40] and included in BiDEL for usability's sake. To summarize, BiDEL is the DEL $\mathcal{L}_C$ with:

$$\mathcal{L}_C = \left\{ \begin{array}{llll} \texttt{Add}_{table}, & \texttt{Del}_{table}, & \texttt{Add}_{column}, & \texttt{Del}_{column} \\ \texttt{Split}_{column}, & \texttt{Unite}_{column}, & \texttt{Split}_{row}, & \texttt{Unite}_{row} \\ \texttt{Rename}_{table}, & \texttt{Rename}_{column} & & \end{array} \right\}$$

All BiDEL SMOs require a set Θ of parameters. Let $inst\,(o, D)$ be the set of instances of the SMO o with valid parameters w.r.t. the database D. For instance, the only parameter to remove a table with $\texttt{Del}_{table}(\Theta)$ is the name of an existing table, so $inst\,(\texttt{Del}_{table}\,(\Theta), D) = \{\texttt{Del}_{table}(R) | R \in D\}$. Further, let $inst\,(\mathcal{L}, D) = \bigcup_{o \in \mathcal{L}} inst\,(o, D)$ be the set of all validly parameterized SMO instances of the DEL $\mathcal{L}$. Then, a BiDEL evolution script S for a database D is a sequence of instantiated SMOs with $S \in inst\,(\mathcal{L}_C, D_i)^+$, where D_i is the database after applying the i-th SMO.

In the following, we specify the semantics of all BIDEL SMOs. Table 4.1 summarizes the definition of the semantics based on $\mathcal{L}_{\text{min}}$. The table also shows the SQL-like syntax we propose for the implementation of BIDEL. In the remainder, $R.C = \{c_1, \ldots, c_n\}$ denotes the set of columns of table R and R_i specifies the revision i of the table R. Whenever an SMO does not change the table's name but its columns or rows, we increment this revision counter i to avoid naming conflicts. BIDEL SMOs take table versions as input and return new table versions. According to the SQL standard, tables are multisets. Our $\mathcal{L}_{\text{min}}$ semantics is based on the relational algebra, so tables are sets. Relational database systems internally manage row identifiers, which are at least unique per table. At the level of SMO implementation, we consider the row identifiers as part of the tables, hence tables are sets. The corresponding multiset semantics of the SMOs can be achieved, by adding a multiset projection of the resulting tables that removes the row identifiers without eliminating duplicates.

$\texttt{Add}_{table}$ and $\texttt{Del}_{table}$: The SMOs $\texttt{Add}_{table}$ and $\texttt{Del}_{table}$ are the simplified version of their $\mathcal{L}_{\text{min}}$ counterparts. $\texttt{Add}_{table}(R, \{c_1, \ldots, c_n\})$ requires two parameters, a table name R and a set of column definitions c_i. It creates an empty table with the specified name and schema. $\texttt{Del}_{table}(R)$ takes only a single parameter, the name of the table to be dropped.

$\texttt{Add}_{column}$ and $\texttt{Del}_{column}$: The SMO $\texttt{Add}_{column}$ adds a new column to an existing table. As parameters, $\texttt{Add}_{column}(R_i, c, f(c_1, \ldots, c_n))$ takes the name R_i of the table where the column should be added, the column definition c of the new column, and a function f. The resulting table is R_{i+1}. $\texttt{Add}_{column}$ applies the function f to each row in R_i to calculate the row's value for the new column c in R_{i+1}. The function f is a standard SQL function that can access the existing values of the row.

$\texttt{Del}_{column}$ removes a column from a table. Specifically, $\texttt{Del}_{column}(R_i, c)$ takes the name R_i of an existing table and the name $c \in R_i.C$ of the column that should be removed from R_i. The resulting table is R_{i+1}. For bidirectional semantics, a function f is required as an additional parameter to calculate the default values for the dropped column according to $\texttt{Add}_{column}$.

$\texttt{Split}_{column}$ and $\texttt{Unite}_{column}$: The SMO $\texttt{Split}_{column}$ splits a table vertically and removes the original table. $\texttt{Split}_{column}$ has a generalized semantics, where the resulting parts are allowed to be incomplete and to overlap. The SMO $\texttt{Split}_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}))$ takes the name R of the original table, a pair consisting out of a table name S and a set of column names s_i as specification of the first partition and optionally a second pair $(T, \{t_1, \ldots, t_m\})$ as specification of the second partition. The two sets of column definitions are independent. In case $S.C \cap T.C \neq \emptyset$, the columns $S.C \cap T.C$ are copied. In case $S.C \cup T.C \subset R.C$, the partitioning is incomplete. If the second partition is not specified, T is not created. BIDEL prohibits empty column sets for S and T since tables must have at least one column.

$\texttt{Unite}_{column}$ is the inverse operation of $\texttt{Split}_{column}$: it joins two tables based on a given condition and removes the original tables. As parameters, $\texttt{Unite}_{column}(R, S, T, cond, o)$ takes the names R and S of the original tables, the name T of the resulting table, a join condition $cond$, and the optional boolean o to indicate an outer join. The join condition is either the primary key, or a specified foreign key, or an arbitrary SQL predicate without further nesting, so N:M mappings are explicitly allowed. The join on the primary key or a foreign key are merely specialized conditions, so the semantics of $\texttt{Unite}_{column}$ is defined for the most general case of an arbitrary condition $cond$. In case $o = \top$, $\texttt{Unite}_{column}$ performs an outer join, so that no rows from the original tables are lost. In case $o = \bot$ (or not specified) $\texttt{Unite}_{column}$ performs an inner join. With the inner join, $\texttt{Unite}_{column}$ loses all rows from R and S that do not find a join partner since R and S are dropped after the join. More complex join types like e.g. left, right, or full outer joins can be simulated in sequences with other SMOs [112]. Note that restricting the join to foreign key relations as other DELs do [41], does not prevent this information loss. A foreign key does not guarantee that every row in the referenced table is actually referenced by at least one row in the referencing table.

Table 4.1: Syntax and semantics of BiDEL operations.

SMO:	$\texttt{Add}_{table}(R, \{c_1, \ldots, c_n\})$
Semantic:	$\texttt{Add}(R, \pi_{c_1,\ldots,c_n}(\emptyset))$;
Syntax:	`CREATE TABLE` R `(`c_1`,...,`c_n`)`
SMO:	$\texttt{Del}_{table}(R)$
Semantic:	$\texttt{Del}(R)$;
Syntax:	`DROP TABLE` R
SMO:	$\texttt{Rename}_{table}(R, R')$
Semantic:	$\texttt{Add}(R', R)$; $\texttt{Del}(R)$;
Syntax:	`RENAME TABLE` R `INTO` R'
SMO:	$\texttt{Rename}_{column}(R_i, c, c')$
Semantic:	$\texttt{Add}\left(R_{i+1}, \rho_{c'/c}(R_i)\right)$; $\texttt{Del}(R_i)$;
Syntax:	`RENAME COLUMN` c `IN` R_i `TO` c'
SMO:	$\texttt{Add}_{column}(R_i, c, f(c_1, \ldots, c_n))$
Semantic:	$\texttt{Add}\left(R_{i+1}, \pi_{R_i.C \cup \{c \leftarrow f(c_1,\ldots,c_n)\}}(R_i)\right)$; $\texttt{Del}(R_i)$;
Syntax:	`ADD COLUMN` c `AS` f`(`c_1`,...,`c_n`)` `INTO` R_i
SMO:	$\texttt{Del}_{column}\left(R_i, c, \underline{f(c_1, \ldots, c_n)}\right)$
Semantic:	$\texttt{Add}\left(R_{i+1}, \pi_{R_i.C \setminus \{c\}}(R_i)\right)$; $\texttt{Del}(R_i)$;
Syntax:	`DROP COLUMN` c `FROM` R_i `DEFAULT` f`(`c_1`,...,`c_n`)`
SMO:	$\texttt{Split}_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}), \underline{cond})$
Semantic:	$\texttt{Add}(S, \pi_{s_1,\ldots,s_n}(R))$; [$\texttt{Add}(T, \pi_{t_1,\ldots,t_m}(R))$]; $\texttt{Del}(R)$;
Syntax:	`DECOMPOSE TABLE` R `INTO` S `(`s_1`,...,`s_n`) [,` T `(`t_1`,...,`t_m`)` `ON (PK | FK` fk `|` $cond$`)]`
SMO:	$\texttt{Unite}_{column}(R, S, T, cond, o)$
Semantic:	$o = \bot$: $\texttt{Add}(T, R \bowtie_{cond} S)$; $o = \top$: $\texttt{Add}(T, R \rtimes_{cond} S)$; $\texttt{Del}(R)$; $\texttt{Del}(S)$;
Syntax:	`[OUTER] JOIN TABLE` R`,` S `INTO` T `ON (PK | FK` fk `|` $cond$`)`
SMO:	$\texttt{Split}_{row}(R, (S, cond_S), (T, cond_T))$
Semantic:	$\texttt{Add}(S, \sigma_{cond_S}(R))$; [$\texttt{Add}(T, \sigma_{cond_T}(R))$]; $\texttt{Del}(R)$;
Syntax:	`PARTITION TABLE` R `INTO` S `WITH` $cond_S$ `[,` T `WITH` $cond_T$`]`
SMO:	$\texttt{Unite}_{row}(R, S, T, \underline{cond_S}, \underline{cond_T})$
Semantic:	$\texttt{Add}\left(T, \pi_{R.C \cup \{\omega \rightarrow a_i \mid a_i \in S.C \setminus R.C\}}(R) \cup \pi_{S.C \cup \{\omega \rightarrow a_i \mid a_i \in R.C \setminus S.C\}}(S)\right)$; $\texttt{Del}(R)$; $\texttt{Del}(S)$;
Syntax:	`MERGE TABLE` R `(`$\underline{cond_R}$`),` S `(`$\underline{cond_S}$`)` `INTO` T

Legend: Underlined parameters are only necessary for bidirectional data propagation

$\mathbf{Split}_{row}$ and $\mathbf{Unite}_{row}$: $\texttt{Split}_{row}$ partitions a table horizontally. Like its vertical counter part $\texttt{Split}_{column}$, its semantics is more general than standard horizontal partitioning [28]. The SMO creates at most two partitions out of a given table—with the partitioning allowed to be incomplete and overlapping—and removes the original table. More precisely, $\texttt{Split}_{row}(R, (S, cond_S), (T, cond_T))$ takes the name R of the original table, a pair of table name S and condition $cond_S$ as specification of the first partition and optionally a second pair $(T, cond_T)$ as specification of the second partition. Both conditions $cond_S$ and $cond_T$ are independent. If the original tables contain rows that fulfill neither of the conditions, the resulting partitioning is incomplete. Rows that fulfill both conditions are copied resulting in overlapping partitions. In case both conditions hold for all rows, i.e., $cond_S = \top$ and $cond_T = \top$, both S and T are complete copies of R. Hence, $\texttt{Split}_{row}$ subsumes the functionality of a copy operation that can be found in other DELs. If $cond_T$ is not specified, $\texttt{Split}_{row}$ does not create table T.

$\texttt{Unite}_{row}$ is the inverse operation of $\texttt{Split}_{row}$; it merges two given tables along the row dimension and removes the original tables. As parameters, $\texttt{Unite}_{row}(R, S, T)$ requires the names R and S of the original tables and the name T of the resulting table. The schema of R and S are not required to by equivalent. In case both schemas differ, T contains null values (ω) in the respective cells. $\texttt{Unite}_{row}$ eliminates duplicates in T. In case R and S contain equivalent rows, these rows will show up only once in T.

$\mathbf{Rename}_{table}$ and $\mathbf{Rename}_{column}$: The last two SMOs rename named schema elements. The SMO $\texttt{Rename}_{table}(R, R')$ renames the table with the name R into R'. $\texttt{Rename}_{column}(R_i, c, c')$ renames the column c in table R_i into c', which results in table R_{i+1} as well known from their SQL-DDL counterparts.

Now, we have a precise definition of both the syntax and semantics of BiDEL; summarized in Table 4.1. Basically, BiDEL allows to create, rename, and drop both columns and table but also to split and merge table both along the tuples or the columns. In the next section, we show that these SMOs are powerful enough to cover common evolution scenarios and provide a formal guarantee for relational completeness.

4.3 EVALUATION

After presenting the syntax and the semantics of BiDEL, we will now discuss the initially set objectives. We evaluate BiDEL's practical completeness (Section 4.3.1) as well as its relational completeness (Section 4.3.2) to show that it is an appropriate DEL for multi-schema-version database systems and that developers can express any occurring evolution scenario. In Section 4.3.3, we show how BiDEL's bidirectionality reduces the length and complexity of the code to be written by developers and thereby yields simpler and more robust database evolution support in multi-schema-version database systems.

4.3.1 Practical Completeness

We use the database evolution benchmark from Carlo Curino et al. [40] that is based on actual evolution histories of open source projects. The benchmark requires e.g. that a DEL can model an excerpt of 171 schema versions of Wikimedia in order to be practically complete. We model the benchmark's evolution history completely with BiDEL. BiDEL proved to be capable of providing the database

SMO	#occurences
CREATE TABLE	42
DROP TABLE	10
RENAME TABLE	1
ADD COLUMN	93
DROP COLUMN	20
RENAME COLUMN	37
JOIN	0
DECOMPOSE	4
MERGE	2
PARTITION	0

Figure 4.2: SMOs of Wikimedia schema evolution.

schema in each version exactly according to the benchmark and migrate the data in a meaningful way. Thus, BiDEL is practical complete and feasible to handle such real world scenarios. Figure 4.2 summarizes characteristics of the 209 SMOs long BiDEL evolution history, particularly how often each SMO has been used in total. We account the dominance of simple SMOs like adding and removing both columns and tables mainly to the restricted database evolution support current DBMSes provide. Still, there are more complex evolutions requiring the other SMOs as well. So, there is a need for more sophisticated database evolution support. In general, the characteristics of the evolution with BiDEL is obviously very similar to the same evolution modeled with the also practically complete DEL PRISM [40]. However, there are slight differences, because the respective sets of SMOs differ. For instance the complex evolution from version v06696 to v06710 takes 31 SMOs instead of 92 SMOs due BiDEL's powerful `DECOMPOSE` SMO.

A major difference in the set of SMOs between PRISM and BiDEL is that PRISM relies on the copy operation as starting point for many practically relevant scenarios. After initially copying the data, further SMOs are used to shape the copy according to the intended evolution. In contrast, BiDEL does not offer an explicit copy operation at all. While the copy operation is easy and intuitive for forward evolution, it breaks the backward evolution/propagation of data. Therefore, BiDEL does not provide an explicit copy operation but implicitly allows copying data within other SMOs—namely `PARTITION` and `DECOMPOSE`. These SMOs already adjust the amount of copied information to the intended evolution, which shortens e.g. the evolution of 171 schema versions of Wikimedia from 269 SMOs with PRISM to 209 SMOs with BiDEL. But most importantly, knowing the explicit intention of copying a table is an essential prerequisite e.g. for backward propagation of data and for merging different schema versions, as we will demonstrate in this thesis.

4.3.2 Relational Completeness

To show the relational completeness of BiDEL, we argue that it is at least as powerful as $\mathcal{L}_{\text{min}}$, which is relationally complete by definition (Section 4.1.1). There is always a semantically equivalent expression in BiDEL for any expression in $\mathcal{L}_{\text{min}}$. The $\texttt{Del}\,(R)$ operation from $\mathcal{L}_{\text{min}}$ is trivial, since it is equivalent to BiDEL's $\texttt{Del}_{table}(R)$. On the contrary, $\texttt{Add}\,(R, \epsilon)$ from $\mathcal{L}_{\text{min}}$ is more complex, as ϵ covers the power of the relational algebra. Since both the relational algebra and BiDEL are closed and composable languages, it is reasonable to address each operation of the relational algebra separately. We show that, for each operation from the relational algebra, there is a semantically equivalent sequence of SMOs in BiDEL.

We consider the basic relational algebra [35] and the common extensions outer join and extended projection. The basic relational algebra contains selection, renaming, projection, cross product, as well as union and difference of relations. This set of relational operations covers the whole relational algebra and subsumes further operations like intersection and division [112]. We intentionally exclude extensions such as the transitive closure and sorting—BIDEL is non-recursive and set-based, which proves to be a reasonable trade-off between expressiveness and usability, however, extending BIDEL's expressiveness with more powerful operations is future research. With respect to implementations based on current database management systems, the distinction between different types of null values is not considered [123]. For instance $\texttt{Unite}_{row}$ adds null values in columns, which existed in only one input table, losing the information, whether a value was null before or did not exist at all. The following sections consider the relational algebra operations [112] plus the chosen extensions and show that BIDEL is capable to obtain the semantically equivalent results.

Relation: R: The basic elements of the relational algebra are relations. They contain the data and are directly accessible by BIDEL as tables. Whenever one table is required multiple times within a relational algebra expression, BIDEL allows to copy them using $\texttt{Split}_{row}(R, (S, \top), (T, \top))$.

Selection: $\sigma_{cond}(R)$: The relational selection operation returns the subset of rows from R, where each row satisfies the condition $cond$. The semantic equivalent is directly provided by BIDEL's $\texttt{Split}_{row}$.

Algorithm 1 BIDEL sequence for relational selection.

1: $\texttt{Split}_{row}(R, (S, cond))$;

The $\texttt{Split}_{row}$ operation with only one output table applies the condition $cond$ to the input table and creates a new table including this data. Applying the defined semantics, we obtain the expected relational selection:

$$S \overset{l.1}{=} \underline{\underline{\sigma_{cond}(R)}} \tag{4.1}$$

Rename: $\rho_{c'/c}(R_i)$: Renaming a column is subsumed by the extended projections, however, we include it here for completeness. BIDEL's obvious semantic equivalent according to Table 4.1 is the SMO $\texttt{Rename}_{column}$.

Algorithm 2 BIDEL sequence for renaming a relation.

1: $\texttt{Rename}_{column}(R_i, c, c')$;

$$R_{i+1} \overset{l.1}{=} \underline{\underline{\rho_{c'/c}(R_i)}} \tag{4.2}$$

Extended Projection: $\pi_P(R_i)$: We use the extended projection since it subsumes the traditional projection. The extended projection defines a new set of columns, whose values are computed by functions depending on the existing columns. The projection $P = \{f_k(R_i.C) \to a_k | 1 \leq k \leq m\}$ produces a relation with m columns, each being computed by a function $f_k(R_i.C)$ taking the $n = |R_i.C|$ columns from R_i as input. For instance, we can project the `Task` table in the schema version TasKy to $\pi_{task \to task, (prio==1) \to isUrgent}(Task)$, so we return the task and a boolean stating whether its priority is 1 or not. The following algorithm describes the BIDEL sequence for the relational projection operation in general.

Algorithm 3 BIDEL sequence for relational projection

1: **for** $k = [1..m]$ **do**
2: $\quad \texttt{Add}_{column}(R_{i+k-1}, a'_k, f_k\,(r_1, \ldots, r_n))$;
3: **for** $r_j \in R_i.C$ **do**
4: $\quad \texttt{Del}_{column}(R_{i+m+j-1}, r_j)$;
5: **for** $k = [1..m]$ **do**
6: $\quad \texttt{Rename}_{column}(R_{i+m+n+k-1}, a'_k, a_k)$;

Without loss of generality, we use for-loops to iterate over the attribute sets. Since this is only schema depending and data independent, it does not extend the expressiveness of the DEL but is simply a short notation. The first SMO adds a new column, with a masked name, for each column of the output table. This allows to compute the new values based on all existing ones. Afterwards, we drop the old columns, rename the new columns to their unmasked name, and remove all intermediate tables. In our example above, we first add the two columns $task'$ and $isUrgent'$ and compute the projected values. Afterwards, we remove the original columns $name$, $prio$, and $author$ and finally rename the projected columns to $task$ and $isUrgent$. Applying the semantics definitions of the BIDEL SMOs to Algorithm 3 results in the desired extended projection, as we will show now. The respectively applied line of the BIDEL sequence (Algorithm 3) is indicated above the equal signs.

$$R_{i+1} \overset{l.2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1}\,(R_i) \tag{4.3}$$

$$R_{i+m} \overset{l.1,2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}\,(R_i) \tag{4.4}$$

$$R_{i+m+1} \overset{l.4}{=} \pi_{r_2,\ldots,r_n,a'_1,\ldots,a'_m}\,(R_{i+m}) \tag{4.5}$$

$$R_{i+m+n} \overset{l.3,4}{=} \pi_{a'_1,\ldots,a'_m}\,(R_{i+m}) = \pi_{f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}\,(R_i) \tag{4.6}$$

$$R_{i+m+n+1} \overset{l.6}{=} \pi_{a'_1\to a_1,a'_2,\ldots,a'_m}\,(R_{i+m+n}) \tag{4.7}$$

$$R_{i+m+n+m} \overset{l.5,6}{=} \pi_{a'_1\to a_1,\ldots,a'_m\to a_m}\,(R_{i+m+n}) = \underline{\underline{\pi_{f_1(R_i.C)\to a_1,\ldots,f_m(R_i.C)\to a_m}\,(R_i)}} \tag{4.8}$$

In Equation 4.3, we apply Line 2 from Algorithm 3 for the first projected column a'_1, which merely adds the calculated value and the column name to the projection clause. According to the loop on Lines 1 and 2 we do this for all projected columns as shown in Equation 4.4. In the second loop on Lines 3 and 4, we drop the original columns of the table since they are no longer required for the calculation. Equation 4.5 shows the result after dropping the first column, while Equation 4.6 shows the table after full iteration of the second loop. Finally, we replace the masked names with the originally intended names of the projected columns on Lines 5 and 6 to first obtain Equation 4.7 and finally the extended projection in Equation 4.8.

Outer Join: $R \rtimes\!\ltimes_p S$**:** The outer join is a common extension to the traditional relational algebra. Beyond the rows according to an inner join, it also includes those rows in the result, which did not find a join partner. The missing values for columns of the other table are filled with null values. BIDEL's $\texttt{Unite}_{column}(R, S, T, p, \top)$ explicitly offers outer joins, which is semantically equivalent.

Algorithm 4 BIDEL sequence for relational outer join operation

1: $\texttt{Unite}_{column}(R, S, T, cond, \bot)$;

$$T \overset{l.1}{=} \underline{\underline{R \rtimes\!\ltimes S}} \tag{4.9}$$

Cross Product: $R \times S$**:** The cross product produces a row in the output table for each pair of rows from the input tables. Algorithm 5 describes the BiDEL sequence to obtain the relational cross product. We add a new column j to both tables with $j \notin R_i.C$ and $j \notin S_k.C$. The default value of j is a constant 1, so we can perform an inner join on j, such that there will be one row in the output table for each pair of rows from the two input tables. Finally, we remove the additional column j and summarize the outcome to the cross product of R and S. Thereby, we show the semantic equivalence between the relational cross product and the presented sequence of BiDEL SMOs.

Algorithm 5 BiDEL sequence for relational cross product

1: $\texttt{Add}_{column}(R_i, j, 1)$;
2: $\texttt{Add}_{column}(S_k, j, 1)$;
3: $\texttt{Unite}_{column}(R_{i+1}, S_{k+1}, T_0, R_{i+1}.j = S_{k+1}.j, \bot)$;
4: $\texttt{Del}_{column}(T_0, j)$;

$$R_{i+1} \overset{l.1}{=} \pi_{r_1,\ldots,r_n,1\to j}\,(R_i) = \{(r_1,\ldots,r_n,1)\,|\,(r_1,\ldots,r_n) \in R_i\} \tag{4.10}$$

$$S_{k+1} \overset{l.2}{=} \{(s_1,\ldots,s_m,1)\,|\,(s_1,\ldots,s_m) \in S_k\} \tag{4.11}$$

$$\begin{aligned} T_0 &\overset{l.3}{=} R_{i+1} \bowtie_{R_{i+1}.j=S_{k+1}.j} S_{k+1} \\ &= \{(r_1,\ldots,r_n,s_1,\ldots,s_m,1)\,|\,(r_1,\ldots,r_n) \in R_i, (s_1,\ldots,s_m) \in S_k\} \end{aligned} \tag{4.12}$$

$$\begin{aligned} T_1 &\overset{l.4}{=} \{(r_1,\ldots,r_n,s_1,\ldots,s_m)\,|\,(r_1,\ldots,r_n) \in R_i, (s_1,\ldots,s_m) \in S_k\} \\ &= \underline{\underline{R \times S}} \end{aligned} \tag{4.13}$$

Union: $R \cup S$**:** The relational union operation merges the rows from both input tables to the one output table including an elimination of duplicates. Using the SMO $\texttt{Unite}_{row}$, BiDEL provides a semantic equivalent to the relational union operation.

Algorithm 6 BiDEL sequence for relational union operation

1: $\texttt{Unite}_{row}(R, S, T)$;

$$T \overset{l.1}{=} \pi_{R.C}\,(R) \cup \pi_{S.C}\,(S) = \underline{\underline{R \cup S}} \tag{4.14}$$

Please note that the union in the relational algebra requires R and S to have identical sets of attributes ($R.C = S.C$), which justifies the simplification step.

Difference: $R \setminus S$**:** The relational difference returns all rows, which occur in the first, but not in the second table. Analogous to the union, it requires R and S to have identical sets of columns ($R.C = S.C$). Algorithm 7 describes the BiDEL sequence to obtain the relational difference, where ω denotes a null value. We add a new column j to S_k with $j \notin S_k.C$ and the default value 1. The outer join on all column pairs in $\{(R_i.c_i, S_k.c_i) | c_i \in R_i.C\}$ (remember $R_i.C = S_k.C$), results in a table containing all rows which were in at least one of the two input tables. However, all rows that occurred in S_k have the value 1 in the column j and are removed by the third SMO. All rows which occurred exclusively in R have a null value ω in the column j and remain as a result. Applying the semantics definition of the SMOs finally leads to the relational difference operation. Please note that $(r_1,\ldots,r_n) \notin S_k$ is equal to $(r_1,\ldots,r_n,1) \notin S_{k+1}$ due to the first step.

Algorithm 7 BiDEL sequence for relational difference operation

1: $\texttt{Add}_{column}(S_k, j, 1)$;
2: $\texttt{Unite}_{column}(R_i, S_{k+1}, T_0, (R_i.c_1 = S_{k+1}.c_1 \wedge \ldots \wedge R_i.c_n = S_{k+1}.c_n), \top)$;
3: $\texttt{Split}_{row}(T_0, (T_1, j \neq \omega))$;
4: $\texttt{Del}_{column}(T_1, j)$;

$$S_{k+1} \overset{l.1}{=} \pi_{s_1,\ldots,s_m,1\to j}(S_k) \tag{4.15}$$

$$\begin{aligned} T_0 &\overset{l.2}{=} R_i \bowtie S_{k+1} \\ &= \{(r_1,\ldots,r_n,1) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \in S_{k+1}\} \\ &\quad \cup \{(r_1,\ldots,r_n,1) \mid (r_1,\ldots,r_n) \notin R_i, (r_1,\ldots,r_n,1) \in S_{k+1}\} \\ &\quad \cup \{(r_1,\ldots,r_n,\omega) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \notin S_{k+1}\} \end{aligned} \tag{4.16}$$

$$\begin{aligned} T_1 &\overset{l.3}{=} \sigma_{\neg(j\neq\omega)}(T_0) = \sigma_{(j=\omega)}(T_0) \\ &= \{(r_1,\ldots,r_n,\omega) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \notin S_{k+1}\} \end{aligned} \tag{4.17}$$

$$\begin{aligned} T_2 &\overset{l.4}{=} \pi_{R.C}(T_1) \\ &= \{(r_1,\ldots,r_n) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n) \notin S_k\} = \underline{\underline{R_i \setminus S_k}} \end{aligned} \tag{4.18}$$

In Equation 4.15, we apply Line 1 from Algorithm 7 so that S_{k+1} contains the added column j with the value 1. Equation 4.16 shows the joint table T_0 according to the outer equi join on all columns except j, such that T_0 is basically a duplicate-eliminating union of R and S, where any tuple occurring is S has $j = 1$ (Line 2). In Equation 4.17, we remove those tuples from S (Line 3), so we can drop the column j in Equation 4.18 (Line 4) and finally end up with the relational difference operation.

Finally, we successfully showed that BiDEL provides a semantic equivalent for each relational algebra expression, which makes it equally expressive as $\mathcal{L}_{\text{min}}$. Hence, BiDEL is a relational complete DEL and a sound foundation for InVerDa and future research.

4.3.3 Bidirectionality

Using the bidirectional DEL BiDEL releases developers from the expensive and error-prone task of manually writing delta code. We show this using both the TasKy example (Figure 2.4, page 28). We implement the evolution from TasKy to TasKy2 with handwritten and hand-optimized SQL and compare this code to the semantically equivalent BiDEL statements. The general setup is a multi-schema-version database system, so both TasKy and TasKy2 should co-exist at the same time and the physical materialization will be migrated from TasKy to TasKy2 eventually. Specifically, we manually implement (1) creating the initial TasKy schema, (2) creating the additional schema version TasKy2 with the respective views and triggers, and (3) migrating the physical table schema to TasKy2 and adapting all existing delta code. This handwritten SQL code is much longer and much more complex than achieving the same goal with BiDEL. Table 4.2 shows the lines of code (LOC) required with SQL and BiDEL, respectively, as well as the ratio between these values. As there is no general coding style for SQL, LOC is a rather vague measure. We also include the objective number of statements and number of characters (consecutive white-space characters counted as one) to get a clear picture and to use the number of characters per statement as an indicator for the code's complexity. Obviously, creating the initial schema in the database is equally complex for both approaches. However, evolving it to the new schema version TasKy2 and migrating the data accordingly requires 359 and 182 lines

InVerDa	Initially	Evolution	Migration
Lines of Code	1	3	1
Statements	1	3	1
Characters	54	152	19
SQL (Ratio)			
Lines of Code	1 (×1.00)	359 (×119.67)	182 (×182.00)
Statements	1 (×1.00)	148 (×49.33)	79 (×79.00)
Characters	54 (×1.00)	9477 (×62.35)	4229 (×222.58)

Table 4.2: Ratio between SQL and InVerDa delta code.

of SQL code respectively, while we can express the same with 3 and 1 lines with BiDEL. Moreover, the SQL code is also more complex, as indicated by the average number of characters per statement. While BiDEL is working exclusively on the visible schema versions, with handwritten SQL developers also have to manage auxiliary tables, triggers etc.—working on multiple technical levels also increases the code's complexity and makes the manual solutions error-prone and expensive.

In sum, we have shown that multi-schema-version database systems with an SMO-based DEL, such as BiDEL, greatly release the developers and DBAs from a lot of manual implementation effort. SMOs carry enough information to generate delta code for access propagation and migration between schema versions automatically, which saves orders of magnitude of manual implementations.

4.4 SUMMARY

We designed and presented BiDEL—the practically complete, relationally complete, and bidirectional DEL of InVerDa. BiDEL is an SMO-based DEL that couples the evolution of both the schema and the data into intuitive evolution steps, which is an essential requirement to let the multi-schema-version database system automatically manage the multitude of schema versions. BiDEL provides a set of SMOs with a comprehensible SQL-like syntax. Further, we defined the semantics of all BiDEL SMOs based on the relational algebra. We showed that BiDEL's SMOs are practically complete—it is possible to model the evolution history of 171 Wikimedia schema versions solely with BiDEL SMOs, which thereby prove to cover practical scenarios. The semantics definition with relational algebra expressions also allowed us to formally evaluate the relational completeness of BiDEL: We defined a minimal relational complete DEL $\mathcal{L}_{\text{min}}$ and showed that BiDEL is at least as expressive as $\mathcal{L}_{\text{min}}$.

Finally, we propose an extended set of arguments for BiDEL's SMOs to make them bidirectional. The pure form of BiDEL describes the forward evolution, and thereby only the forward propagation of data. In addition, the bidirectional extensions require the developers to specify strategies for the backward propagation of data for those SMOs where the backward propagation is not intuitively obvious from the forward evolution. We have evaluated that the implementation effort for developers and DBAs to realize co-existing schema versions can be reduced by orders of magnitude thanks to the rich semantics of BiDEL. In sum, BiDEL proved to be a feasible DEL to specify the evolution in a multi-schema-version database system as we can give important completeness guarantees and BiDEL is powerful enough to automatically generate delta code for data propagation or data migration both backwards and forwards between schema versions.

MERGING EXISTING SCHEMA VERSIONS WITH MIX

BiDEL is a powerful DEL that couples the evolution of both the schema and the data in simple SMOs and thereby captures the developers' intentions. This facilitates valuable simplifications and supporting tools for database developers. In this chapter, we present a **semi-automatic consolidation algorithm**, called MIX, which takes two existing schema versions and creates a new schema version by merging them. MIX comes in handy, e.g. when different development teams evolve the database schema in different branches and want to consolidate the respective evolutions to one consolidated schema version afterwards. Traditional SQL hides the developers' intentions between the lines, namely between DDL and DML and DQL statements. In contrast, SMO-based DELs, such as BiDEL, represents the intention explicitly, which essentially allows us to semi-automatically consolidate the intentions of multiple evolutions in the first place. In Section 5.1, we discuss the objectives posed on such a functionality. The overall idea of merging two schema versions is based on a local consolidation of every pair of SMOs from the two evolutions as we describe in Section 5.2. Therefore, we consider every possible combination of two SMOs and specify the intuitively expected result of merging these two SMOs in a consolidation matrix as presented in Section 5.3. We extend this local consolidation of pairs of SMOs to the consolidation of whole sequences of SMOs in Section 5.4, which allows the developers to easily create new schema versions. Finally, we conclude this chapter in Section 5.5.

5.1 OBJECTIVES

The objective of the consolidation algorithm is to merge the intentions of two schema versions into a new schema version. The two existing schema versions have a common—potentially empty—schema version from where their evolutions continued independently. Given the two schema versions to merge, the consolidation algorithm should create a new schema version that represents the intentions of the evolutions from the last common schema version to these two schema versions as close as possible. Since these evolutions are given as sequences of SMOs, the developers' intentions are kept explicitly, which facilitates to semi-automate the consolidation algorithm.

Obviously, the consolidated sequence of SMOs would need to start at one of the two given schema versions. Creating it as a merged version that depends on both given schema versions would create a cycle in the version history, which is by design acyclic to keep the propagation of any data access deterministic. When issuing the merge operation, the first mentioned schema version should be the core schema and the intention of the second mentioned schema version should be incorporated as close as possible. Generally speaking, there is an evolving core database schema and a derived variant. The variant has been first defined on the initial version of the core and should be semi-automatically mapped to an evolved version of the core. Merging the intention of one schema version into another schema version is closely related to the co-evolution problem of database schemas.

Given a sequence of SMOs S_V that evolves the core database D^{Core} to a variant $D^{Variant}$, and another sequence of SMOs $S_{C'}$ that evolves the core database D^{Core} to a new version $D^{Core'}$, the goal is to propose a sequence $S_{V'}$ that evolves $D^{Core'}$ to the co-evolved variant $D^{Variant'}$ and preferably maintains the intentions of S_V in $S_{V'}$. Thereby, the consolidation algorithm should **consolidate the intentions of $\mathbf{S_{V'}}$ with the intention of $\mathbf{S_{C'}}$**. To summarize the problem of variant co-evolution:

$$Given: \quad D^{Core} \xrightarrow{S_V} D^{Variant} \text{ and } D^{Core} \xrightarrow{S_{C'}} D^{Core'}$$
$$Goal: \quad D^{Core'} \xrightarrow{S_{V'}} D^{Variant'}$$

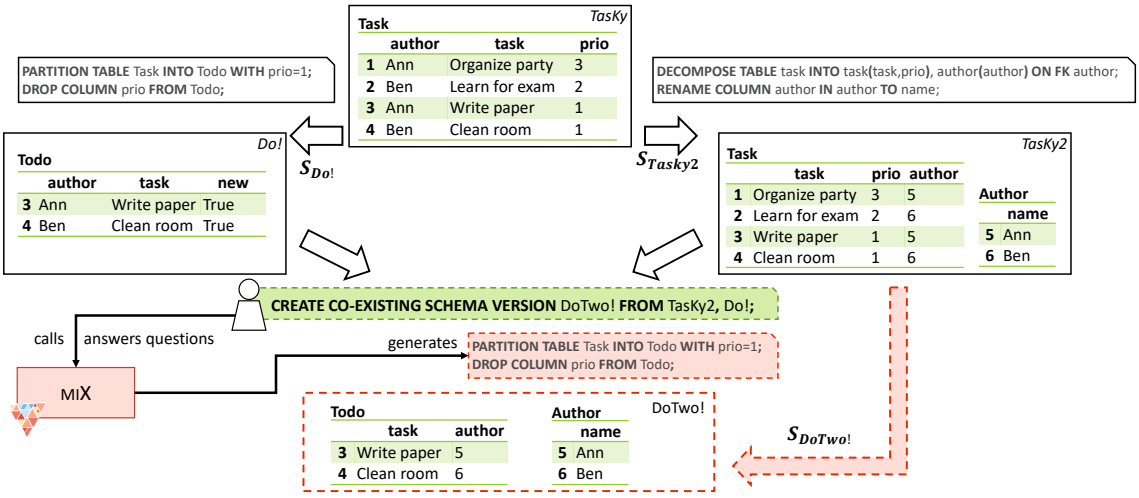


Figure 5.1: Merging two branches of the TasKy example.

Let us clarify the envisioned consolidation algorithm using the TasKy example (Figure 2.4, page 28) with the two different evolution to Do! and TasKy2 both starting at the initial schema version TasKy. The initial schema version TasKy is obviously the core D^{Core}. The variant $D^{Variant}$ of this core is Do!. The variant Do! should be merged into the core evolution $S_{C'}$, which creates the schema version TasKy2. The merged schema version $D^{Variant}$ that starts at TasKy and incorporates the intentions of Do! is called DoTwo! in our example. Figure 5.1 illustrates how merging the two different evolutions should work: When the developers create a new schema version by merging two existing ones, the consolidation algorithm semi-automatically generates a sequence of SMOs $S_{DoTwo!}$ for the evolution from TasKy to DoTwo! that incorporates the evolution $S_{Do!}$ as close as possible. Intuitively, we expect the `Task` table version in TasKy2 to be evolved to a `Todo` table with only tasks of the highest priority 1. Further, the priority column should be dropped as well. The consolidation algorithm should merely map the `PARTITION` SMO to the `Task` table of TasKy2. The same applies to the `RENAME COLUMN` SMO. This whole process should be automated as much as possible. As there are situations that cannot be foreseen when developing the two given schema versions, it should be at least semi-automatic: During the execution of the consolidation algorithm, the developers need to clarify ambiguous situations whenever they occur. In our example, everything works nicely without further interaction. However, assuming that e.g. the `DECOMPOSE` SMO replicates the `prio`column to both target table versions, the selection criteria of the `PARTITION` SMO would be applicable to both target table versions of the `DECOMPOSE` SMO. In this case, the consolidation algorithm should ask the developers, to which target table version(s) the `PARTITION` SMO should be propagated. The asked questions should be very simple, to make the merging schema versions as easy and robust as possible.

5.2 SMO CONSOLIDATION

We present MIX, the consolidation algorithm for merging schema versions in INVERDA. MIX follows a two level algorithm: globally consolidating sequences of SMOs by locally consolidating pairs of SMOs. In this section, we present the local level, where MIX combines one SMO of the variant with one SMO of the core evolution. This combination process is backed by a *consolidation matrix* that defines the intuitively expected result for any pair of SMOs (Section 5.3). On the global level, MIX's consolidation algorithm extends the local consolidation of two single SMOs to two sequences of SMOs, which finally allows semi-automatic variant co-evolution as we show in Section 5.4.

Combining two BiDEL SMOs from $inst\left(\mathcal{L}_C, D^{Core}\right)$, one from the core and one from the variant, is the very foundation of MIX. Therefore, we use the SMO composition operation $\circ$, which is defined as $\circ : inst\left(\mathcal{L}_C, D^{Core}\right) \times inst\left(\mathcal{L}_C, D^{Core}\right) \rightarrow inst\left(\mathcal{L}_C, D^{Core}\right)^*$ and takes a core SMO_{core} and a variant SMO_{var} to return a sequence SMO^* that can be executed after the SMO_{core}. Executing the derived sequence of SMOs after SMO_{core} should represent the intention of SMO_{var} as close as possible. The result of $\circ$ is a sequence of SMOs since one SMO_{var} may result in multiple SMOs after the consolidation. For instance, the core partitions a table and the variant's evolution has to be applied to both partitions.

According to our objective, the composition operation $\circ$ is not commutative. As shown in Figure 5.1, the core evolution remains unchanged, thus the composition distinguished between core and variant SMOs. The co-evolved variant SMOs S'_V have to start at the evolved core $Core'$. On the application level, this allows to keep the evolved core application unchanged but requires corresponding evolution of the variant application. Hence, the SMO consolidation $\circ$ is not commutative to ensure that SMO_{Core} remains unchanged and the derived sequence SMO^* is applicable after this SMO_{Core}.

The result of the SMO composition $\circ$ is the intuitively expected combination of the two input SMOs. However, the intuitive expectation is hard to grasp formally. Hence, MIX stores these intuitive expectations explicitly: the consolidation matrix represents the consolidation operation $\circ$ including the intuitively expected outcome for each possible pair of SMOs. MIX can handle any consolidation matrix matching the interface of the composition operation $\circ$. In this work, we propose one possible consolidation matrix for the general case, but MIX is by no means limited to it.

Based on the example in Figure 5.1, let us consider e.g. the consolidation of the core's DECOMPOSE with the variant's PARTITION SMO. In general, this pair can be consolidated only semi-automatically—the developers need to specify to which target tables of the core's DECOMPOSE SMO the variants partitioning should be propagated:

$$\begin{aligned}
&\texttt{Split}_{column}(T_1, (S_1, \{A_1\}), (S_2, \{A_2\}), cond) \circ \texttt{Split}_{row}(T_1, (R_1, cond_1), (R_2, cond_2)) \\
&= \begin{cases}
(\texttt{Split}_{row}(S_1, (R_1?, cond_1), (R_2?, cond_2))) & Option\ 1 \\
(\texttt{Split}_{row}(S_2, (R_1?, cond_1), (R_2?, cond_2))) & Option\ 2 \\
(\texttt{Split}_{row}(S_1, (R_1?, cond_1), (R_2?, cond_2)), & Option\ 3 \\
\quad \texttt{Split}_{row}(S_2, (R_1?, cond_1), (R_2?, cond_2))) &
\end{cases}
\end{aligned}$$

MIX will propose to use the same names for the two out coming tables of the partitioning as before, however, the developers can assign other names if intended (symbolized by e.g. $R_1?$, which asks for an alternative name of R_1). In our specific TasKy example, things are easier since there is only one target table version of the DECOMPOSE SMO that allows to evaluate the partitioning condition, which makes it obvious for MIX to choose $Option\ 1$: So, we end up with the intuitively expected schema, where the partitioned table is decomposed just as the unpartitioned one before:

$$\begin{aligned}
&\texttt{Split}_{column}(Task, (Task, \{task, prio\}), (Author, \{author\}), FK(fk_{author})) \\
&\circ \texttt{Split}_{row}(Task, (Todos, prio = 1)) \\
&= \texttt{Split}_{row}(Task, (Todos, prio = 1)))
\end{aligned}$$

In sum, the consolidation of two SMOs is realized with the $\circ$ operation that takes two SMOs and applies the intention of the second one as good as possible to the first one. Since, this might involve ambiguities like e.g. naming conflicts, we keep the developers in the loop to answer very simple questions. Thanks to the well-defined interface of $\circ$, MIX clearly separates the definition of the intuitive expectations from the effective consolidation algorithm.

Table 5.1: Compatibility matrix for SMO consolidation.

	Variant SMO									
Core SMO	$\texttt{Add}_{table}$	$\texttt{Del}_{table}$	$\texttt{Rename}_{table}$	$\texttt{Split}_{row}$	$\texttt{Unite}_{row}$	$\texttt{Split}_{column}$	$\texttt{Unite}_{column}$	$\texttt{Add}_{column}$	$\texttt{Del}_{column}$	$\texttt{Rename}_{column}$
$\texttt{Add}_{table}$	$?_N$	✓	$?_N$	✓	✓	✓	✓	✓	✓	✓
$\texttt{Del}_{table}$	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
$\texttt{Rename}_{table}$	$?_N$	✓	$?_N$	✓	✓	✓	✓	✓	✓	✓
$\texttt{Split}_{row}$	✓	✓	$?_N$	✓	$?_P$	✓	$?_P$	✓	✓	✓
$\texttt{Unite}_{row}$	✓	✗	$?_N$	✓	✓	$?_E$	✓	✓	✓	✓
$\texttt{Split}_{column}$	✓	✓	$?_N$	$?_P$	✓	✗	✓	$?_P$	✓	✓
$\texttt{Unite}_{column}$	✓	✗	$?_N$	✓	✓	$?_E$	✓	✓	✓	✓
$\texttt{Add}_{column}$	✓	✓	✓	✓	✓	$?_E$	✓	$?_N$	✓	$?_N$
$\texttt{Del}_{column}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\texttt{Rename}_{column}$	✓	✓	✓	✓	✓	✓	✓	$?_N$	✓	$?_N$

5.3 CONSOLIDATION MATRIX

We propose a consolidation matrix containing each possible combination of SMO_{core} and SMO_{var} and the intuitively expected output for the general case. In specific domains it seems to be reasonable to adapt this matrix accordingly. However, already the generally proposed co-evolved variant SMOs greatly simplify the co-evolution and can be individually exchanged or extended when needed. As SMOs are compact and intuitive, developers can easily adapt the proposed sequence of SMOs and tweak it to the concretely expected variant.

Table 5.1 shows the compatibility of all combinations of a core SMO and a variant SMO. Most pairs can be consolidated automatically (✓, 71 out of 100 possible combinations). Some combinations of SMOs are contradicting (✗, 11 out of 100 possible combinations), hence the respective SMO_{var} cannot be executed at all—which might be just fine in many scenarios—so the resulting consolidated schema might still match the intention of the variant evolution reasonably. Further, there are three different kinds of user interaction (?, 18 out of 100 possible combinations). First, the developers have to resolve naming conflicts ($?_N$, 12 times). Second, the developers have to choose out of two or three possible paths for a consolidation when the core splits a table ($?_P$, 3 times). And finally, the developers have to extend the list of columns of an SMO_{var}=$\texttt{Split}_{column}$ in case there are newly created columns in the core ($?_E$, 3 times). All questions are comprehensible and easy to answer by selecting an option or typing a name, which significantly simplifies the developers' effort for the variant co-evolution. In the following, we will explain the consolidation matrix in more detail. First, we discuss how to consolidate SMOs on table level (SMO_{table}) with any other SMO. Afterwards, we go through the consolidation matrix line by line to elaborate on the consolidation of an SMO_{core} both on row (SMO_{row}) and on column (SMO_{column}) level with any other SMO_{var}.

SMO$_{table}$ ∘ **SMO**$_{table}$**:** Consolidating SMOs that work on different input tables is trivial, as they do not interfere at all, so a core's or a variant's `Add`$_{table}$ cannot conflict with any other SMO. The only exception are naming conflicts—if such a naming conflict occurs, MIX asks the developers to provide a new name for the variant's table. Any combination of `Add`$_{table}$ SMOs and `Rename`$_{table}$ SMOs might require the developers to assign a new name to the variant table in case it should have the same name as a core table. Apart from this, `Add`$_{table}$ never has overlapping input with another SMO. If the variant removes a table that is used by SMO_{core}, MIX can easily propagate this to the target tables of SMO_{core} and delete those tables as well, e.g. if the variant intents to drop the same table as the core drops, this intention is implicitly covered. However, if the variant's developers intent to rename a table that has been removed in the core evolution, this intention cannot be reflected anymore and will be ignored. In the remainder of the consolidation matrix, we will focus on those cases where the SMOs have an overlap in the set of input tables.

SMO$_{table}$∘**SMO**$_{row,column}$**:** Any table that has been added in the core evolution cannot be the input of any SMO_{var}, since it does not exist in the variant in the first place, which makes the consolidation trivial. If the core deletes a table that also serves as input for SMO_{var}, the consolidation will ignore SMO_{var} as the concept was intentionally removed from the database in the core evolution. Finally, a renamed table in the core evolution can simply be renamed in SMO_{var} as well.

SMO$_{row,column}$ ∘ **SMO**$_{table}$**:** Whenever the SMO_{var} creates a new table, this will obviously not conflict with any other SMO_{core} since the created table does not exist in the core schema at all. Given the variant's developers intent to drop a table which has been evolved by the core developers, MIX simply drops all tables resulting from the respective SMO_{core}. However, this does not apply when SMO_{core} is either `Split`$_{column}$ or `Split`$_{row}$ on the table to drop: In these cases, we ignore the variant's SMO `Del`$_{table}$, as the variant needs to keep the data of the other input table. Given a `Rename`$_{table}$ in the variant evolution, we can only consolidate it straight forward with the core evolution, if the latter does not create any new tables, which holds for adding, removing, and renaming columns only. For all other SMO_{core} that split or unite tables, we ask the developers to give names for the newly created tables as well.

SMO$_{row}$ ∘ **SMO**$_{row,column}$**:** The two tables resulting from a core's `Split`$_{row}$ have the same structure as the input table, which allows propagating any SMO_{var} simply to both resulting tables as well. Merely when consolidating a core's `Split`$_{row}$ with a variant SMO that has two input tables (`Unite`$_{row}$ and `Unite`$_{column}$), the developers need to decide to which output of the SMO_{core} to propagate the SMO_{var}. It is not possible to propagate them to both output tables of SMO_{core}, as the respective other input table of SMO_{var} does exist only once. Further, a core's `Unite`$_{row}$ produces one output table whose columns are the union of the two input tables. Hence, any SMO_{var} working on one input table does also work on the output table and can be propagated naïvely. The only SMO_{var} to take special care of is `Split`$_{column}$. We offer the developers to extend the column sets of the output tables of `Split`$_{column}$ as the input table from the updated core may have more columns, now.

Split$_{column}$ ∘ **SMO**$_{row,column}$**:** Finally, we discuss the core SMOs on column level starting with focusing on the core's `Split`$_{column}$. A variant's `Split`$_{row}$ defines a condition on the set of attributes, which results in three potential scenarios: First, if the condition can be evaluated on only one resulting table of SMO_{core}, then the SMO_{var} is applied to the respective table only. Second, if the condition cannot be evaluated on any resulting table, MIX has to ask the developers to specify a new condition or it ignores the SMO_{var}. Third, if the condition can be evaluated on both resulting tables, MIX asks the developers whether to propagate the SMO_{var} to the first, to the second, or to both resulting tables. When the variant applies a `Unite`$_{row}$, the developers have to choose which resulting table of the core's `Split`$_{column}$ to merge within the variant evolution. Again, this cannot be applied to both resulting tables since the merge-partner exists only once. Given the SMO_{core} is `Split`$_{column}$,

it is not possible to also apply a variant's $\texttt{Split}_{column}$ since the original set of column might be incompletely/redundantly distributed over the two output tables. Hence the original concept which should be decomposed in the variant does not exist any longer. If the variant intents joining a table which has been decomposed by the core, the variant's join is applied to that resulting table of the core's evolution that allows evaluating the join condition—if this holds for both, the developers have to choose and if it does not hold for any resulting table, MIX will simply ignore SMO_{var}. Finally, MIX can propagate an $\texttt{Add}_{column}$ through a core's $\texttt{Split}_{column}$ by explicitly asking the developers to which output table(s) to propagate the new column—dropping and renaming a column is always propagated to the resulting table that contains the respective column.

$\mathbf{Unite}_{column} \circ \mathbf{SMO}_{row,column}$: When the core joins two tables ($\texttt{Unite}_{column}$), any SMO_{var} operating on one of those tables can be applied to the joined table since all columns of the original table are reflected in the joined one as well. Only if SMO_{var} is $\texttt{Split}_{column}$, the developers are offered to extend the column sets of the decomposition in order to incorporate the columns of the joined table.

$\{\mathbf{Add}_{column}, \mathbf{Del}_{column}, \mathbf{Rename}_{column}\} \circ \mathbf{SMO}_{row,column}$: Given the SMO_{core} extends the set of columns by adding a new one with $\texttt{Add}_{column}$ and the variant decomposes the same table with $\texttt{Split}_{column}$. The developers need to adjust the column set of the variants $\texttt{Split}_{column}$. Basically, they have to decide whether the new column shall be added to the first, the second, both, or none of the two tables resulting from the decomposition. Further, an added column in the core evolution can cause naming conflicts, if SMO_{var} adds or renames a column that should have the same name—MIX asks the developers to adjust the name in the SMO_{var}. In all other cases, an added column in the core evolution does not affect any SMO_{var} since all relevant columns are definitively still available. In general, if any naming conflict occurs with $\texttt{Add}_{column}$ or $\texttt{Rename}_{column}$, the developers have to resolve this ambiguity by merely providing a new name for the column in SMO_{var}. Deleting a column in the core evolution is only crucial if the variant SMO works on the respective column. Particularly, the dropped column can simply be removed from variant's $\texttt{Split}_{column}$ and MIX prompts the developers to specify a new condition if (a) the partitioning criteria of $\texttt{Split}_{row}$ or (b) the join condition of $\texttt{Unite}_{column}$ or (c) the function calculating a new value for $\texttt{Add}_{column}$ cannot be evaluated any longer. In all other cases, the dropped column does not affect the variants SMO. Finally, renaming a column in the core evolution simply requires MIX to automatically rename the respective column within all subsequent variant SMOs.

Summing up, we can consolidate most pairs of SMOs automatically. There are destructive SMO_{core} that remove either tables or columns, which renders certain SMO_{var} impossible. Nevertheless, this can still meet the developers' intentions. Furthermore, there are a few pairs that require the developers' interaction. The questions that MIX asks the developers are short and simple, keeping the whole semi-automatic co-evolution process convenient. Mainly, we ask the developers to clarify naming conflicts, extend the column sets of $\texttt{Split}_{column}$, or to choose output tables of splitting SMOs. The proposed consolidation results meet the intuitive expectations and should cover the general case.

5.4 EVOLUTION CONSOLIDATION

Both the variant and the core evolution are defined with sequences of SMOs S_V and $S_{C'}$. To co-evolve the variant and obtain $S_{V'}$, we extend the SMO composition operation $\circ$ from single SMOs to sequences of SMOs. The sequence composition $\bullet : S_V \times S_{C'} \rightarrow SMO^*_{var'}$ returns the sequence of SMOs $S_{V'}$ starting at the new core version. To consolidate whole sequences of SMOs, the general idea is to repeatedly apply the SMO composition $\circ$. Figuratively speaking, MIX takes each variant SMO from S_V and propagates each one through the core evolution $S_{C'}$ while applying the SMO composition $\circ$ at every single step.

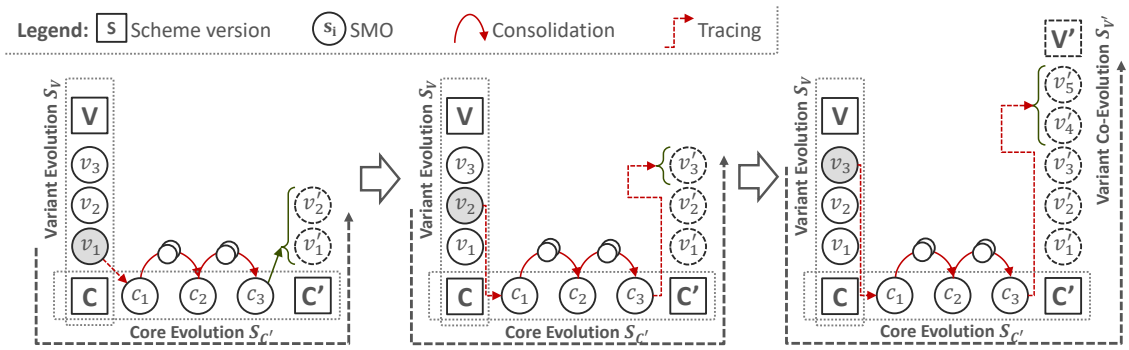


Figure 5.2: Consolidation of sequences of SMOs.

The algorithm for the co-evolution process of a variant V created by $C \xrightarrow{S_V} V$ with the core evolution $C \xrightarrow{S_{C'}} C'$, is illustrated in Figure 5.2. The final result is the co-evolved variant evolution $C' \xrightarrow{S_{V'}} V'$. In each depicted step, MIX propagates one variant SMO from S_V through all core SMOs in $S_{C'}$. For each pair of SMOs, MIX first checks, whether their sets of input tables overlap. If not, MIX can simply proceed with the next core SMO. If yes, MIX applies the composition operation $\circ$ presented before. For the first SMO v_1 in Figure 5.2 this works just fine and MIX appends the resulting SMOs to $S_{V'}$.

Already in the second step, we cannot immediately propagate the variant SMO v_2 through the core evolution since it potentially works on different tables. The first variant SMO v_1 might e.g. rename a table. To compensate for such evolutions, MIX adapts v_2 to represent the intended evolution on the tables of the initial core schema. This tracing process needs to be inverted after propagating the respective SMO_{var} through the core evolution to obtain the correct $S_{V'}$. MIX continues until all variant SMOs are propagated through the core evolution and appended to $S_{V'}$.

Mapping variant SMOs to the evolved core table versions is not always trivially possible. Particularly, this tracing of the tables gets broken by those SMOs that create new tables. The subsequent SMOs cannot be mapped to core tables naïvely. The best-effort solution to this problem is to immediately jump to the co-evolved variant and apply the subsequent SMOs of the original variant right there. This is usually possible as long as all required columns survived the core evolution.

Finally, one remark on the escalation of contradicting SMOs: Whenever MIX cannot consolidate a variant SMO s, because s contradicts the core evolution (marked with ✗ in Table 5.1), all subsequent variant SMOs that rely on s cannot be consolidated with the core as well. When e.g. the variant adds a column to a table, however, the core evolution intends to drop this table, MIX has to ignore the variant SMO since the core evolution $S_{C'}$ must not be changed. This also affects all subsequent variant SMOs that rely on the added column in the variant. As the core concepts, on which the variant was based, are removed by the core evolution, the developers have to rethink this part of the variant either way. Pragmatically, MIX simply ignores the variant SMO and leaves it to the developers to integrate the contradicting variant.

Let us revisit the example in Figure 5.1 on page 73. The `PARTITION` SMO on the `Task` table is the first SMO of the variant evolution from TasKy to Do!. When propagating it through the `DECOMPOSE` SMO of the core evolution from TasKy to TasKy2, MIX simply applies the partitioning to the target table `Task` since the condition requires the `prio` column, which is not available in the `Author` table. If the `prio` column would have been replicated to both target tables of the `DECOMPOSE` SMO, MIX would have asked the developers to which target table or target tables that `PARTITIONING` SMO should be propagated. Since this is not the case, MIX merely asks the developers to confirm or change the names of the target tables. In our example, the developers confirm the names, so MIX can further propagate the `PARTITION` SMO through the core evolution's `RENAME COLUMN` SMO, which is trivial.

The final SMO of the variant is the `DROP COLUMN` of the column `prio` in `Task`, which can be consolidated directly with the core's `DECOMPOSE` and `RENAME COLUMN` SMO. There are neither naming conflicts nor any other ambiguities, so the `DROP COLUMN` SMO is merely appended to the final sequence of SMO that evolves the evolved core schema version TasKy2 to the co-evolved schema version DoTwo! as shown in Figure 5.1. We obtain the co-evolved variant as expected and MIX proposes it to the developers without any further interaction required.

As a summary, the global consolidation algorithm of MIX takes two sequences of SMOs and merges their intentions. By propagating each SMO of the variant through every SMO of the core evolution, a sequence of co-evolved variant SMOs is derived. This sequence describes an evolution from the evolved core to the co-evolved variant. MIX's consolidation algorithm relies on the local consolidation of pairs of SMOs: In each step, one SMO of the variant is consolidated with one SMO of the core according to the presented consolidation matrix. Thereby, the consolidation algorithm allows to easily create new schema versions in the multi-schema-version database system by merging two existing schema versions.

5.5 SUMMARY

Summing up, we presented MIX, a tool in INVERDA that uses the rich semantics of BIDEL's SMOs to semi-automatically create new schema versions by merging the intention of one schema version (variant) into the intention of another schema version (core). MIX's consolidation algorithm utilizes the strength of BIDEL: Developers provide all the relevant information explicitly when creating the new core or variant schema version, which facilitates MIX to release developers from the repetitive task of applying core-evolutions manually to the respective variants. We have introduced a consolidation matrix that contains the expected result for consolidating any pair of two SMOs. By extending this consolidation of two single SMOs to the consolidation of sequences of SMOs, MIX's consolidation algorithm can create the new merged schema version. To resolve contradictions or ambiguities of different intents, we choose a semi-automatic approach that keeps the developers in the loop and clarifies such conflicts by posing very simple questions to the developers. The semi-automatic consolidation algorithm of MIX releases the developers from manually merging N evolution steps of the core evolution with M evolution steps of the variant evolution, which entails $N \times M$ pairs that need to be consolidated and checked for conflicts. MIX helps avoiding faulty evolutions and allows the developers to focus on the actual implementation task, which increases the agility and quality of database development significantly.

DATA INDEPENDENCE FOR CO-EXISTING SCHEMA VERSIONS

Multi-schema-version database systems allow to create and manage co-existing schema versions. This is basically already possible with database version management tools like e.g. Liquibase, but to the best of our knowledge, all existing tools are limited to exactly one accessible schema version per database; there is currently no support to have truly co-existing schema versions within one database that all access the same conceptual data set. Multi-schema-version database systems, as we envision them, allow all schema versions to co-exist at the same time within the same database. Write operations in any schema version are immediately visible in all other schema versions as well—but each single schema version behaves like a full-fledged single-schema database, such that applications do not notice that their schema version is part of a multi-schema-version database system. The prerequisite for this functionality is that developers use a bidirectional SMO-based DEL to describe the evolution in the first place. These SMOs describe how to propagate any data access both forwards and backwards between schema versions and facilitate data independence, hence the possibility to independently change the physical materialization without affecting the accessibility of any schema version.

In this chapter, we will first discuss the specific objectives of full data independence in a multi-schema-version database system in Section 6.1. In Section 6.2, we define the semantics for propagating data through SMOs both forwards and backwards, which particularly includes the identification and management of the auxiliary information that needs to be stored additionally in order to persist the data of all non-materialized table versions. Using these semantics, we describe the generation of delta code for co-existing schema versions in Section 6.3. In Section 6.4, we formally evaluate that, in fact, each single schema version behaves like a regular single-schema database, no matter in which table versions the data is actually materialized. This is an important guarantee that allows to safely use multi-schema-version database systems without the risk of corrupting the data. In Section 6.5 we conduct an empiric evaluation of co-existing schema versions in multi-schema-version database systems focusing on the performance gains facilitated by the independent physical materialization. Finally, we summarize the chapter in Section 6.6.

6.1 OBJECTIVES

Having multiple co-existing schema versions within the same database raises the challenge to store the data for all these schema versions efficiently. Obviously, one possibility is to materialize each table version physically. This *fully redundant* approach is the easiest to implement as each application can read the physical table versions just as usual and write operations are propagated to all other table versions according to the SMOs' bidirectional semantics (Chapter 4). Apart from this, the fully redundant materialization is not very appealing, since a lot of storage capacity is required and write operations are propagated to all replicas, which makes updates more expensive. The contrary possibility is to materialize merely a *non-redundant* subset of the table versions; data accesses to non-materialized table versions need to be propagated through the SMOs to the materialized table versions. Since not all SMOs are information preserving, we would not only propagate data accesses to other table versions according to the bidirectional semantics of the SMOs but also to auxiliary tables that store the otherwise lost information. Therefore, the bidirectional mapping semantics should be extended to **materialization-independent mapping semantics** that additionally cover the persistence of auxiliary information, if the respective table versions are not materialized. Based on such materialization-independent mapping semantics, we could easily create any degree of *partially redundant* materializations for a multi-schema-version database system where each schema version is guaranteed to behave like a regular single-schema database. In this chapter, we focus on the non-redundant materialization and formally evaluate the data independence. Non-redundant materialization is the most difficult materialization; with additionally materialized table versions in partially or fully redundant materializations, we merely leave out the management of the auxiliary information and directly access the materialized table versions.

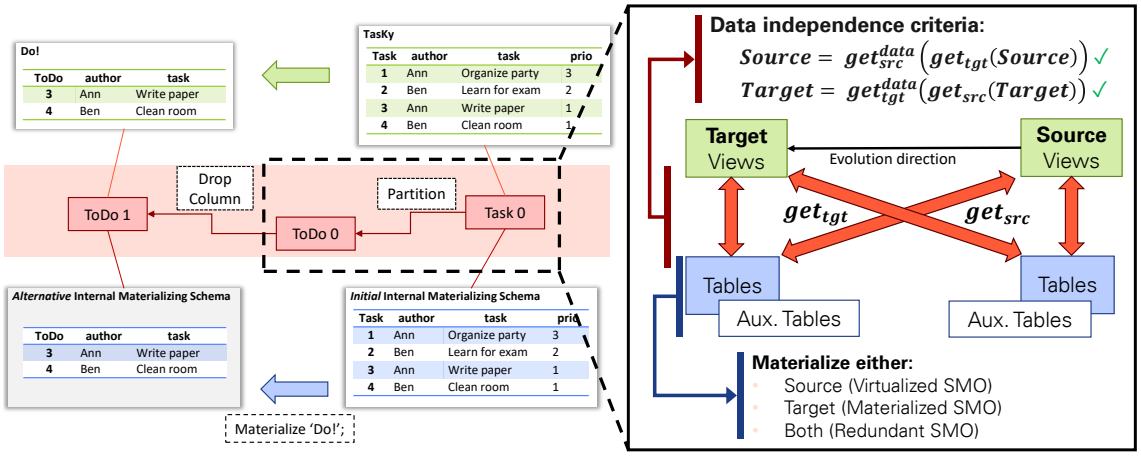


Figure 6.1: Data independence criteria in the TasKy example.

To achieve the objective of non-redundant materialization of multiple co-existing schema versions, the multi-schema-version database system should provide full data independence. Drilling down the data independence to single SMOs, means that both the source and the target side of an SMO should be full-fledged database schemas that can be read and written as any common single-schema database. Particularly, data that has been written in one schema version should be readable unchanged in the same schema version again, even if the respective schema version is not physically materialized. The data should be primarily stored at one side of the SMO. Using the bidirectional semantics of BiDEL's SMOs for generating delta code, allows to propagate the data between schema versions according to the developers' intentions. Data independence of bidirectional SMOs requires that no matter on which side of the SMO the data is physically stored, both sides should behave like a common single-schema database.

Storing data non-redundantly at one side of an SMO is a challenge since most SMOs are not information preserving. To this end, multi-schema-version database systems should store the otherwise lost information in auxiliary tables with each SMO. Figure 6.1 zooms into one specific SMO; namely into the partitioning of the tasks in our TasKy example, which we will use as a running example for the remainder of this chapter. As can be seen, each SMO has a source schema and a target schema—it should be possible that only one of them actually stores the data tables and auxiliary tables physically. Data written at one side of the SMO, which is stored at the opposite side and read back to its origin side, should survive this round trip without any information loss. To formally evaluate this data independence of SMOs, we have to consider three cases depending on the SMO's materialization state: data is either (1) stored on the target side (SMO is materialized) or (2) stored on the source side (SMO is virtualized) or (3) stored on both sides (SMO is redundant). In all cases, data written in any version should be correctly readable in the same version again without any information lost or gained, which requires the correct propagation from and to the data tables and auxiliary tables through the SMO if the respective version is not materialized.

The processing of read and write operations at the materialized side is trivial as data is directly accessed without any schema transformation. To read and write data at the unmaterialized side, the bidirectional SMO semantics comes into play. Let's start with the first case (1); the data is materialized on the target side. For a correct source-side propagation, the data D_{src} at the source side should be mapped by the mapping function γ_{trg} to the data tables and auxiliary tables at the target side (write) and mapped back by the mapping function γ_{src} to the data tables on the source side (read) without any loss or gain visible in the data tables at the source side. Similar conditions have already

been defined for symmetric relational lenses [65]—given data at the source side, storing it at the target side, and mapping it back to source should return the identical data at the source side. For the second case (2), the same conditions should hold vice versa. The third case (3) is trivial, because data can always be read locally and no auxiliary tables are required. Formally, the data independence of an SMO requires the following two conditions to hold:

$$D_{src} = \gamma_{src}^{data}(\gamma_{trg}(D_{src})) \tag{6.1}$$

$$D_{trg} = \gamma_{trg}^{data}(\gamma_{src}(D_{trg})) \tag{6.2}$$

Data tables that are visible to the user need to match these data independence conditions. As indicated by the index γ^{data}, we project away potentially created auxiliary tables. These auxiliary tables are always empty except for SMOs that calculate new values: e.g. adding a column requires to store the calculated values when data is stored at the source side to ensure repeatable reads.

The materialized side of the SMO does not necessarily need to exist physically in its whole; while the auxiliary tables need to be stored physically, the data tables can be provided virtually by another SMO in the same spirit. This concept allows to create sequences of SMOs—a.k.a. evolutions—where the data is stored non-redundantly according to the table versions of one specific slice of the evolution history. All other table versions should be provided virtually using the bidirectional semantics of our BiDEL SMOs. In sum, DEL's with SMOs satisfying the discussed data independence criteria (Equation 6.1 and 6.2) allow co-existing schema versions within the same database where each single schema version behaves like a regular single-schema database. The SMOs of a multi-schema-version database system should guarantee this data independence.

6.2 DATA INDEPENDENT MAPPING SEMANTICS

BiDEL's unique feature is the bidirectional and materialization independent semantics of its SMOs, which is the basis for InVerDa's co-existing schema versions with flexible materialization. We highlight the design principles behind BiDEL SMOs and formally validate their data independence. All BiDEL SMOs follow the same design principles. Without loss of generality, the `PARTITION` SMO is used as a representative example in this section to explain the concepts. The remaining SMOs are introduced in Appendix A.

Figure 6.2 shows the principle structure of a single SMO instance resulting from the sample statement

`PARTITION TABLE` T **`INTO`** R **`WITH`** c_R**`,`** S **`WITH`** c_S

which horizontally splits a source table T into two target tables R and S based on conditions c_R and c_S. If all the table versions are materialized, then reads and writes on both schema versions can be simply delegated to the corresponding data tables T_D, R_D, and S_D, respectively. However, we aim for a non-redundant materialization at one side of the SMO instance, only. If the data is physically stored on the source side of an SMO instance, the SMO instance is called *virtualized*; with data stored on the target side it is called *materialized*. In any case, reads and writes on the unmaterialized side are mapped to the materialized side.

The semantics of each SMO is defined by the two functions γ_{trg} and γ_{src} that precisely describe the mapping from the source side to the target side and vice versa. Assuming the target side of `PARTITION` is materialized, all reads on T are mapped by γ_{src} to reads on R_D and S_D; and writes on T are mapped by γ_{trg} to writes on R_D and S_D. While the payload data of R, S, and T is stored in the physical

tables R_D, S_D, and T_D, the tables R^-, S^+, S^-, R^*, S^*, and T' are auxiliary tables for the PARTITION SMO to prevent information loss. Recall that the semantics of SMOs is data independent, if reads and writes on both the source and the target schema work correctly regardless on which of both sides the data is physically stored. This basically means that each schema version acts like a full-fledged database schema; however, it does not enforce that data written in any version is also fully readable in other versions. In fact, BiDEL ensures this for all SMOs except of those that create redundancy—in these cases the developers specify a preferred replica beforehand.

Obviously, there are different ways of defining γ_{trg} and γ_{src}; in this thesis, we propose one way that systematically covers all potential inconsistencies and guarantees data independence. We aim at a non-redundant materialization, which also includes that the auxiliary tables merely store the minimal set of required auxiliary information. Starting from the basic semantics of the SMO—e.g. the horizontal partitioning of a table—we incrementally detect inconsistencies that contradict the data independence or the bidirectional semantics and introduce respective auxiliary tables. The proposed rule sets can serve as a blueprint since they clearly outline which information need to be stored to achieve data independence. Potential other approaches might be e.g. more compact, but at the end of the day they have to store the same information.

To define γ_{trg} and γ_{src}, we use Datalog—a compact and solid formalism that facilitates both a formal evaluation of data independence and easy delta code generation. Precisely, we use Datalog rule templates instantiated with the parameters of an SMO instance. For brevity of presentation, we use some extensions to the standard Datalog syntax: For variables, small letters represent single attributes and capital letters lists of attributes. For equality predicates on attribute lists, both lists need to have the same length and same content, i.e. for $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, $A = B$ holds if $n = m \wedge a_1 = b_1 \wedge \ldots \wedge a_n = b_n$. All tables have an attribute p, which is an InVerDa-managed identifier to uniquely identify tuples across versions. Additionally, p ensures that the multiset semantics of a relational database fits with the set semantics of Datalog, as the unique key p prevents equal tuples in one relation. For a table T we assume $T(p, \_)$ and $\neg T(p, \_)$ to be safe predicates since any table can be projected to its key.

For the exemplary PARTITION, let's assume this SMO instance is materialized, i.e. data is stored on the target side, and let's consider the γ_{trg} mapping function first. PARTITION horizontally splits a table T from the source schema into two tables R and S in the target schema based on conditions c_R and c_S:

$$R(p, A) \leftarrow T(p, A), c_R(A) \tag{6.3}$$

$$S(p, A) \leftarrow T(p, A), c_S(A) \tag{6.4}$$

The conditions c_R and c_S can be arbitrarily set by the user so that Rule 6.3 and Rule 6.4 are insufficient wrt. the desired materialization-independent semantics, since the source table T may contain tuples neither captured by c_R nor by c_S. In order to avoid inconsistencies and ensure data independence for the SMO, we store the uncovered tuples with $c_S(A) \vee c_R(A) = \bot$ in the auxiliary table T' on the target side:

$$T'(p, A) \leftarrow T(p, A), \neg c_R(A), \neg c_S(A) \tag{6.5}$$

When we materialize the PARTITION SMO, we do no longer store the table version T but the partitions R and S as well as the auxiliary table T'.

Let's now consider the γ_{src} mapping function for reconstructing T while the target side is still considered to be materialized. Reconstructing T from the target side is essentially a union of R, S, and T'. Nevertheless, c_R and c_S are not necessarily disjoint, so there are tuples satisfying the condition $c_S(A) \wedge c_R(A) = \top$. One source tuple may occur as two equal but independent instances in R and S. We call such two instances *twins*. Twins can be updated independently resulting in *separated twins*,

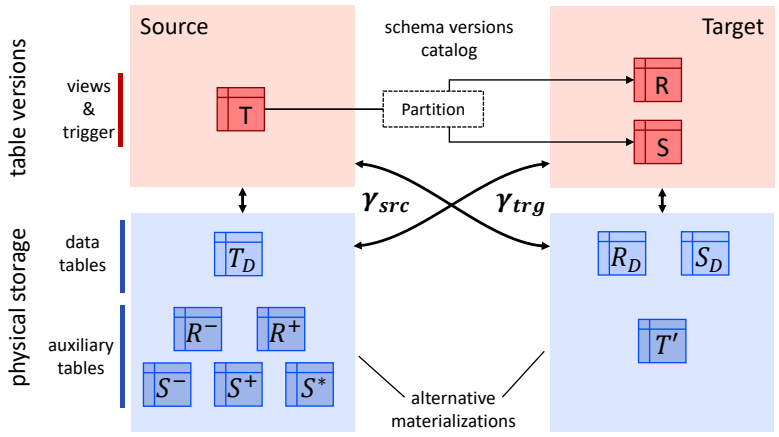


Figure 6.2: Mapping functions of single PARTITION SMO.

i.e. two tuples—one in R and one in S—with equal key p but different value for the other attributes. To resolve this ambiguity and ensure data independence, we consider the first twin in R to be the primus inter pares and define γ_{src} of PARTITION to propagate back all tuples in R as well as those tuples in S that are not contained in R:

$$T(p, A) \leftarrow R(p, A) \tag{6.6}$$
$$T(p, A) \leftarrow S(p, A), \neg R(p, \_) \tag{6.7}$$
$$T(p, A) \leftarrow T'(p, A) \tag{6.8}$$

The Rules 6.3–6.8 define sufficient semantics for PARTITION as long as the target side is materialized.

Let's now assume the SMO instance is virtualized, i.e. data is stored on the source side, and let's keep considering the γ_{src} mapping function. Again, R and S can contain separated twins—unequal tuples with equal key p. According to Rule 6.7, T stores only the separated twin from R. To avoid losing the other twin in S, it is stored in the auxiliary table S^+:

$$S^+(p, A) \leftarrow S(p, A), R(p, A'), A \neq A' \tag{6.9}$$

Accordingly, γ_{trg} has to reconstruct the separated twin in S from S^+ instead of T (concerns Rule 6.4). Twins can also be deleted independently resulting in a *lost twin*. Given the data is materialized on the source side, a lost twin would be directly recreated from its other twin via T. To avoid this information gain and keep lost twins lost, γ_{src} keeps the keys of lost twins from R and S in auxiliary tables R^- and S^-, respectively:

$$R^-(p) \leftarrow S(p, A), \neg R(p, \_), c_R(A) \tag{6.10}$$
$$S^-(p) \leftarrow R(p, A), \neg S(p, \_), c_S(A) \tag{6.11}$$

Accordingly, γ_{trg} has to exclude lost twins stored in R^- from R (concerns Rule 6.3) and those in S^- from S (concerns Rule 6.4). Twins result from data changes issued to the target schema containing R and S which can also lead to tuples that do not meet the conditions c_R resp. c_S. In order to ensure that the reconstruction of such tuples is possible from a materialized table T, auxiliary tables R^* and S^* are employed for identifying those tuples using their identifiers (concerns Rules 6.3 and 6.4).

$$S^*(p) \leftarrow S(p, A), \neg c_S(A) \tag{6.12}$$
$$R^*(p) \leftarrow R(p, A), \neg c_R(A) \tag{6.13}$$

The full rule sets of γ_{trg} respectively γ_{src} are now bidirectional and ensure data independence:

$$\gamma_{\mathbf{trg}}:$$

$$R(p,A) \leftarrow T(p,A), c_R(A), \neg R^-(p) \tag{6.14}$$
$$R(p,A) \leftarrow T(p,A), R^*(p) \tag{6.15}$$
$$S(p,A) \leftarrow T(p,A), c_S(A), \neg S^-(p), \neg S^+(p,\_) \tag{6.16}$$
$$S(p,A) \leftarrow S^+(p,A) \tag{6.17}$$
$$S(p,A) \leftarrow T(p,A), S^*(p), \neg S^+(p,\_) \tag{6.18}$$
$$T'(p,A) \leftarrow T(p,A), \neg c_R(A), \neg c_S(A), \neg R^*(p), \neg S^*(p) \tag{6.19}$$

$$\gamma_{\mathbf{src}}:$$

$$T(p,A) \leftarrow R(p,A) \tag{6.20}$$
$$T(p,A) \leftarrow S(p,A), \neg R(p,\_) \tag{6.21}$$
$$T(p,A) \leftarrow T'(p,A) \tag{6.22}$$
$$R^-(p) \leftarrow S(p,A), \neg R(p,\_), c_R(A) \tag{6.23}$$
$$R^*(p) \leftarrow R(p,A), \neg c_R(A) \tag{6.24}$$
$$S^+(p,A) \leftarrow S(p,A), R(p,A'), A \neq A' \tag{6.25}$$
$$S^-(p) \leftarrow R(p,A), \neg S(p,\_), c_S(A) \tag{6.26}$$
$$S^*(p) \leftarrow S(p,A), \neg c_S(A) \tag{6.27}$$

In sum, these bidirectional mapping semantics of the PARTITION SMO allow to propagate data both forwards and backwards through the SMO and to store auxiliary information if required to correctly persist the data of a non-materialized table version. The semantics of all other BiDEL SMOs is defined in a similar way. For brevity, we do not elaborate on all of them here and recommend the interested reader to consult Appendix A. This precise definition of BiDEL's SMOs, is the basis for the automated delta code generation as well as the formal validation of their data independence.

6.3 DELTA CODE GENERATION

To make a schema version available, InVerDa translates the γ_{src} and γ_{trg} mapping functions into delta code—specifically views and triggers. Views implement delta code for reading; triggers implement delta code for writing. In a genealogy of schema versions, a single table version is the target of one SMO instance and the source for a number of SMO instances. The delta code for a specific table version depends on the materialization state of the table's adjacent SMOs, i.e. it depends on where the data is physically stored.

If both the source and the target side of an SMO are materialized (SMO is redundant), the delta code generation is trivial since no auxiliary tables are needed. Specifically, we remove all rules that have an auxiliary table as head or contain auxiliary tables as positive literals; in all other rules, we remove literals of negated auxiliary tables, which basically leaves us with the regular bidirectional mapping semantics of BiDEL's SMOs for redundant materialization. To determine the right rule sets for delta code generation for non-redundant materialization, consider the exemplary evolution in Figure 6.3. Schema version T_i is materialized, hence the two subsequent SMO instances, $i-1$ and i store their data at the target side (materialized), while the two subsequent SMO instances, $i+1$ and $i+2$ are set to source-side materialization (virtualized). Without loss of generality, three cases for delta code generation can be distinguished, depending on the direction a specific table version needs to go for to reach the materialized data.

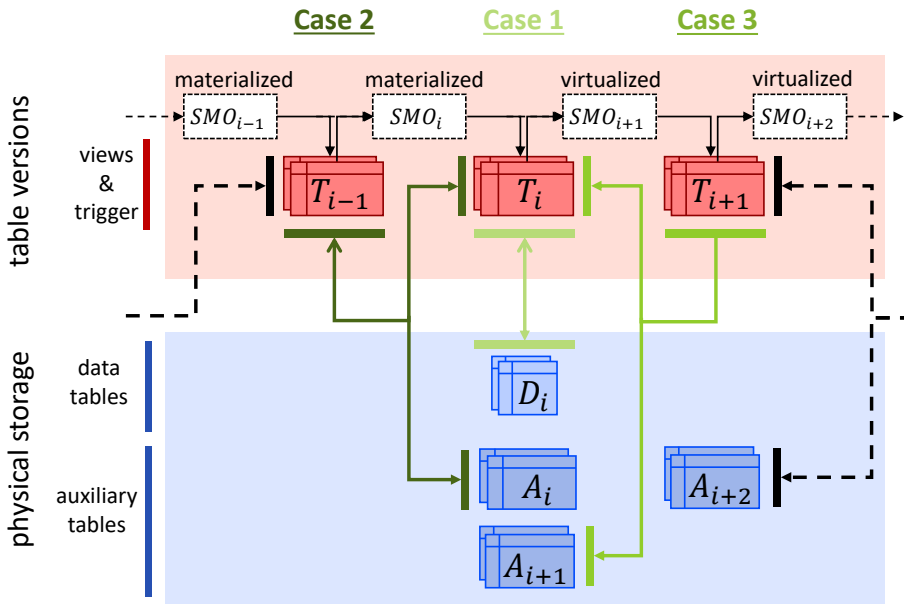


Figure 6.3: Three different cases in delta code generation.

Case 1 – local: The incoming SMO is materialized and all outgoing SMOs are virtualized. The data of the table versions in T_i is stored in the corresponding data table D_i and is directly accessible. The view and the trigger that implement T_i directly propagate reads and writes to data table D_i without any schema translation.

Case 2 – forwards: The incoming SMO and one outgoing SMO are materialized. The data of T_{i-1} is stored in newer versions in the schema versions catalog, so data accesses are propagated with γ_{src} (read) and γ_{trg} (write) of $\texttt{SMO}_i$. The view and the triggers implementing T_{i-1} propagate data along the materialized outgoing $\texttt{SMO}_i$ to views and auxiliary tables of T_i.

Case 3 – backwards: The incoming SMO and all outgoing SMOs are virtualized. The data of table version in T_{i+1} is stored in older versions in the schema versions catalog, so data access is propagated with γ_{trg} (read) and γ_{src} (write) of $\texttt{SMO}_{i+1}$. The view and the triggers that implement T_{i+1} propagate backwards along the virtualized incoming $\texttt{SMO}_{i+1}$ to views and auxiliary tables of version T_i.

In Case 1 delta code generation is trivial. In Case 2 and 3, InVerDa essentially translates the Datalog rules defining the relevant mapping functions into view and trigger definitions. Figure 6.4 illustrates the general pattern of the translation of Datalog rules to a view definition. As a single table can be derived by multiple rules, e.g. Rule 6.20–6.22, a view is a union of subqueries each representing one of the rules. For each subquery, InVerDa lists all attributes of the rule head in the select-clause. Within a nested subselect these attributes are either projected from the respective table version or derived by a function (e.g. values for an added column). All positive literals referring to other table versions or auxiliary tables are listed in the from-clause. Further, InVerDa adds join conditions to the where-clause for each attribute that occurs in multiple positive literals. Finally, the where-clause is completed with conditions, such as $c_S(X)$, and negative literals, which InVerDa adds as a `NOT EXISTS(<subselect for the literal>)`.

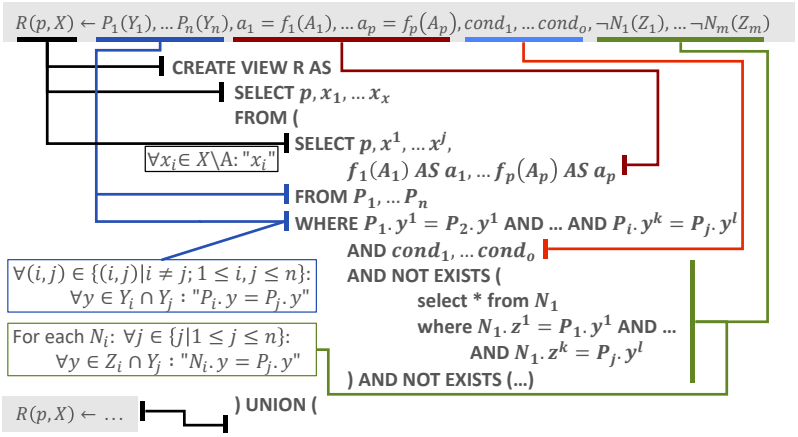


Figure 6.4: SQL generation from Datalog rules.

The generated delta code for writing, are three triggers on each table version—for inserts, deletes, and updates respectively. To not recompute the whole data of the materialized side whenever a write operation is executed at the not-materialized side of an SMO, InVerDa adopts an update propagation technique for Datalog rules [19] that results in minimal write operations. For instance, an insert operation $\Delta_T^+(p, A)$ on the table version T propagated back to the source side of a materialized `PARTITION` SMO results in the following update rules:

$$\Delta_R^+(p, A) \leftarrow \Delta_T^+(p, A), \texttt{new}\ c_R(A), \texttt{old}\ \neg R(p, A) \tag{6.28}$$

$$\Delta_S^+(p, A) \leftarrow \Delta_T^+(p, A), \texttt{new}\ c_S(A), \texttt{old}\ \neg S(p, A) \tag{6.29}$$

$$\Delta_{T'}^+(p, A) \leftarrow \Delta_T^+(p, A), \texttt{new}\ \neg c_R(A), \neg c_S(A), \texttt{old}\ \neg T'(p, A) \tag{6.30}$$

The propagation can handle multiple write operations at the same time and distinguishes between `old` and `new` data, which represents the state before and after applying the write operations. The derived update rules match the intuitive expectations: The inserted tuple is propagated to R or to S or to T' given it satisfies either c_R or c_S or none of them. The additional conditions on the `old` literals ensures minimality by checking whether the tuple already exists in the respective table. To generate trigger code from the update rules, InVerDa applies essentially the same algorithm as for view generation to determine the sets of tuples that need to be inserted, deleted, or updated.

Writes performed on one table version are propagated by an instead-of trigger further to the neighboring table versions. Thereby the write operations are propagated stepwise along the schema versions history to all other table versions as long as the respective update rules deduce write operations, i.e. as long as some data is physically stored with a table version either in the data table or in auxiliary tables. With the generated delta code, InVerDa propagates writes from any schema version to every other co-existing schema version in the schema versions catalog.

In sum, the bidirectional mapping semantics specified as sets of Datalog rules can be directly used to generate delta code for the propagation of both read and write operations along the schema version history to the physically materialized table versions. We use views for the propagation of read operations and instead-of triggers on these views for write propagation. Hence, data accesses are exclusively handled by standard database artifacts, which ensures that the performance and the transaction guarantees of a common database system are also given in multi-schema-version database systems. Data accesses are propagated step-by-step through each SMO individually, which keeps the whole delta code generation simple and robust.

6.4 FORMAL EVALUATION OF DATA INDEPENDENCE

The Condition 6.1 and 6.2 for data independence are shown by applying and simplifying the rule sets that define the mappings γ_{src} and γ_{trg}. Given any input relation, we apply γ_{src} and γ_{trg} in the order according to the respective condition and compare the outcome to the original relation. They have to be identical in order to satisfy the data independence conditions. As neither the rules for a single SMO nor the schema versions catalog have cycles, there is no recursion at all, which simplifies evaluating the combined Datalog rules. The literals for the input data D_{src} or D_{trg} are labeled to distinguish them from the resulting relation and avoid unintended recursion, so e.g. the literal for a source table R is renamed to R_D.

In the following, we introduce some basic notion about Datalog rules as basis for the formal evaluation. A Datalog rule is a clause of the form $H \leftarrow L_1, \ldots, L_n$ with $n \geq 1$ where H is an atom denoting the rule's head, and $L_1, \ldots, L_n$ are literals, i.e. positive or negative atoms, representing its body. For a given rule r, we use $\texttt{head}(r)$ to denote its head H and $\texttt{body}(r)$ to denote its set of body literals $L_1, \ldots, L_n$. In the mapping rules defining γ_{src} and γ_{trg}, every $\texttt{head}(r)$ is of the form $q^r(p, Y)$ where q^r is the derived predicate, p is the InVerDa-managed identifier, and Y is a potentially empty list of variables. Further, we use $\texttt{pred}(r)$ to refer to the predicate symbol of $\texttt{head}(r)$. For a set of rules $\mathcal{R}$, $\mathcal{R}^q$ is defined as $\{r \mid r \in \mathcal{R} \wedge \texttt{pred}(r) = q\}$. For a body literal L, we use $\texttt{pred}(L)$ to refer to the predicate symbol of L and $\texttt{vars}(L)$ to denote the set of variables occurring in L. In the mapping rules, every literal $L \in \texttt{body}(r)$ is either of the form $q_i^r(p, Y_i^r, X_i^r)$ or of the form $c^r(Y_i^r, X_i^r)$, where $Y_i^r \subset Y$ are the variables occurring in L and $\texttt{head}(r)$ and X_i^r are the variables occurring in L but not in $\texttt{head}(r)$. Generally, we use capital letters to denote multiple variables. For a set of literals $\mathcal{K}$, $\texttt{vars}(\mathcal{K})$ denotes $\bigcup_{L \in \mathcal{K}} \texttt{vars}(L)$. The following lemmas are used for simplifying a given rule set $\mathcal{R}$ into a rule set $\mathcal{R}'$ such that $\mathcal{R}'$ derives the same facts as $\mathcal{R}$.

Lemma 1 (Deduction)**.** *Let $L \equiv q^r(p, Y)$ be a literal in the body of a rule r. For a rule $s \in \mathcal{R}^{pred(L)}$ let $\texttt{rn}(s, L)$ be rule s with all variables occurring in the head of s at positions of Y variables in L be renamed to match the corresponding Y variable and all other variables be renamed to anything not in $\texttt{vars}(\texttt{body}(r))$. If L is*

1. *a positive literal, s can be applied to r*
   to get rule set $r(s)$ of the form $\{\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus \{L\} \cup \texttt{body}(\texttt{rn}(s, L))\}$.

2. *a negative literal, s can be applied to r to get rule set*
   $r(s) = \{\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus \{L\} \cup t(K) \mid K \in \texttt{body}(\texttt{rn}(s, L))\}$
   with either $t(K) = \{\neg q_i^s(p, Y_i^s, \_)\}$ if $K \equiv q_i^s(p, Y_i^s, X_i^s)$
   or $t(K) = \{q_j^s(p, Y_j^s, X_j^s) \mid q_j^s(p, Y_i^s, X_j^s) \in \texttt{body}(\texttt{rn}(s, L)) \wedge X_j^s \cap X_i^s \neq \emptyset\} \cup \{c^r(Y_i^s, X_i^s)\}$
   if $K \equiv c^r(Y_i^s, X_i^s)$.[1]

For a given p, let r be every rule in $\mathcal{R}$ having a literal $L \equiv p(X, Y)$ in its body. Accordingly, $\mathcal{R}$ can be simplified by replacing all rules r and all $s \in \mathcal{R}^p$ with all $r(s)$ applications to $\mathcal{R} \setminus (\{r\} \cup \mathcal{R}^p) \cup (\bigcup_{s \in \mathcal{R}^{pred(L)}} r(s))$.

Figuratively speaking, Lemma 1 applies one rule set to another rule set. For every literal L that is the head of rule s, we replace every occurrence of L in the body of any rule r with the body of rule s, hence we replace the literal $L \in \texttt{body}(r)$ with its definition according to rule s. Whenever this leads to a negated list of literals, which is not allowed in Datalog, we resolve this using De Morgan's law.

[1] Correctness can be shown with help of first order logic.

Lemma 2 (Empty Predicate). *Let $r \in \mathcal{R}$ be a rule, L be a literal in the body $L \in \texttt{body}(r)$ and the relation $\texttt{pred}(L)$ is known to be empty. If L is a positive literal, r can be removed from $\mathcal{R}$. If L is a negative literal, r can be simplified to $\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus \{L\}$.*

Lemma 3 (Tautology). *Let $r, s \in \mathcal{R}$ be rules and L and K be literals in the bodies of r and s, respectively, where r and s are identical except for L and K, i.e. $\texttt{head}(r) = \texttt{head}(s)$ and $\texttt{body}(r) \setminus \{L\} = \texttt{body}(s) \setminus \{K\}$, or can be renamed to be so. If $K \equiv \neg L$, r can be simplified to $\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus \{L\}$ and s can be removed from $\mathcal{R}$.*

Lemma 4 (Contradiction). *Let $r \in \mathcal{R}$ be a rule and L and K be literals in its body $L, K \in \texttt{body}(r)$. If $K \equiv \neg L$, r can be removed from $\mathcal{R}$.*

Lemma 5 (Unique Key). *Let $r \in \mathcal{R}$ be a rule and $q(p, X)$ and $q(p, Y)$ be literals in its body. Since, by definition, p is a unique identifier, r can be modified to $\texttt{head}(r) \leftarrow \texttt{body}(r) \cup \{X = Y\}$.*

Lemma 6 (Spare Literal). *Let $r \in \mathcal{R}$ be a rule and $q(\_, X)$ be literals in its body. If X does not occur in any other literal or in the head $\texttt{head}(r)$, r can be reduced to $\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus q(\_, X)$.*

Lemma 7 (Subsumed Literal). *Let $r \in \mathcal{R}$ be a rule and $q_i(\_, X, Y)$ and $q_j(\_, X, \_)$ be literals in its body. Since the literal $q_j(\_, X, \_)$ is always satisfied when $q_i(\_, X, Y)$ is satisfied, it is subsumed by the latter and r can be modified to $\texttt{head}(r) \leftarrow \texttt{body}(r) \setminus q_j(\_, X, \_)$.*

Lemma 8 (Subsumed Rule). *Let $r_1 \in \mathcal{R}$ and $r_2 \in \mathcal{R}$ be two rules from $\mathcal{R}$ with $r_1 \neq r_2$ that have the same rule head, so $\texttt{head}(r_1) = \texttt{head}(r_2)$. If the body of the rule r_2 is a subset of the body of rule r_2, so $\texttt{body}(r_2) \subseteq \texttt{body}(r_1)$, the rule $r2$ can be removed from $\mathcal{R}$.*

We use these lemmas to show the data independence criteria for the PARTITION SMO in detail. First, we show that Criteria 6.1 holds for the materialized PARTITION SMO in Section 6.4.1. Second, we formally evaluate Criteria 6.2 for the virtualized case in Section 6.4.2 as well.

6.4.1 Data Independence of the Materialized PARTITION SMO

At first, we will show that data written at the unpartitioned source side, but stored at the materialized target side, can be read at the source side again without any information being lost. For target-side materialization, Equation 6.1 needs to be satisfied. Writing data T_D from source to target-side results in the mapping $\gamma_{trg}(T_D)$. With target-side materialization, all source-side auxiliary tables are empty. Thus, $\gamma_{trg}(T_D)$ can be simplified with Lemma 2:

$$R(p, A) \leftarrow T_D(p, A), c_R(A) \tag{6.31}$$
$$S(p, A) \leftarrow T_D(p, A), c_S(A) \tag{6.32}$$
$$T'(p, A) \leftarrow T_D(p, A), \neg c_R(A), \neg c_S(A) \tag{6.33}$$

Reading the source-side data back from R, S, and T' to T adds the rule set γ_{src} (Rule 6.20–6.27) to the mapping. Using Lemma 1, the mapping $\gamma_{src}(\gamma_{trg}(T_D))$ simplifies to:

$$T(p, A) \leftarrow T_D(p, A), c_R(A) \tag{6.34}$$
$$T(p, A) \leftarrow T_D(p, A), c_S(A), \neg T_D(p, A) \tag{6.35}$$
$$T(p, A) \leftarrow T_D(p, A), c_S(A), \neg c_R(A) \tag{6.36}$$
$$T(p, A) \leftarrow T_D(p, A), \neg c_S(A), \neg c_R(A) \tag{6.37}$$
$$R^-(p) \leftarrow T_D(p, A), c_S(A), \neg T_D(p, A), c_R(A) \tag{6.38}$$
$$R^-(p) \leftarrow T_D(p, A), c_S(A), \neg c_R(A), c_R(A) \tag{6.39}$$
$$R^*(p) \leftarrow T_D(p, A), c_R(A), \neg c_R(A) \tag{6.40}$$

$$S^+(p,A) \leftarrow T_D(p,A), c_S(A), T_D(p,A'), c_R(A'), A \neq A' \tag{6.41}$$
$$S^-(p) \leftarrow T_D(p,A), c_R(A), \neg T_D(p,A), c_S(A) \tag{6.42}$$
$$S^-(p) \leftarrow T_D(p,A), c_R(A), \neg c_S(A), c_S(A) \tag{6.43}$$
$$S^*(p) \leftarrow T_D(p,A), c_S(A), \neg c_S(A) \tag{6.44}$$

With Lemma 4, we omit Rule 6.35 as it contains a contradiction. With Lemma 3, we reduce Rules 6.36 and 6.37 to Rule 6.46 by removing the literal $c_S(A)$:

$$T(p,A) \leftarrow T_D(p,A), c_R(A) \tag{6.45}$$
$$T(p,A) \leftarrow T_D(p,A), \neg c_R(A) \tag{6.46}$$

The resulting rules for T can be simplified again with Lemma 3 to the following:

$$T(p,A) \leftarrow T_D(p,A) \quad . \tag{6.47}$$

For Rule 6.41, Lemma 5 implies $A = A'$, so this rule can be removed based on Lemma 4. Likewise, the Rules 6.39–6.44 have contradicting literals on T_D, c_R, and c_S respectively, so that Lemma 4 applies here as well. The result clearly shows that data T_D in D_{src} is mapped by $\gamma_{src}(\gamma_{trg}(D_{src}))$ to the target side and back to D_{src} without any information loss or gain:

$$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{D_{src}})) : \; T(p,A) \leftarrow T_D(p,A) \quad \square \tag{6.48}$$

Thus, $D_{src} = \gamma_{src}(\gamma_{trg}(D_{src}))$ holds. Remember that the auxiliary tables only exist on the materialized side of the SMO (target in this case). Hence, it is correct that there are no rules left producing data for the source-side auxiliary tables.

6.4.2 Data Independence of the Virtualized PARTITION SMO

We will now show the opposite direction, so that Condition 6.2 for source-side materialization holds as well. Writing data R_D and S_D from the target-side to the source-side is done with the mapping $\gamma_{src}(R_D, S_D)$. With source-side materialization all target-side auxiliary tables are not required, so we apply Lemma 2 to obtain:

$$\gamma_{\mathbf{src}}(\mathbf{R_D}, \mathbf{S_D}) :$$
$$T(p,A) \leftarrow R_D(p,A) \tag{6.49}$$
$$T(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_) \tag{6.50}$$
$$R^-(p) \leftarrow S_D(p,A), \neg R_D(p,\_), c_R(A) \tag{6.51}$$
$$R^*(p) \leftarrow R_D(p,A), \neg c_R(A) \tag{6.52}$$
$$S^+(p,A) \leftarrow S_D(p,A), R_D(p,A'), A \neq A' \tag{6.53}$$
$$S^-(p) \leftarrow R_D(p,A), \neg S_D(p,\_), c_S(A) \tag{6.54}$$
$$S^*(p) \leftarrow S_D(p,A), \neg c_S(A) \tag{6.55}$$

Reading the target-side data back from the source-side adds the rule set γ_{trg} (Rule 6.14–6.19) to the mapping. Using Lemma 1, the mapping $\gamma_{trg}(\gamma_{src}(T_D))$ extends to:

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{R_D}, \mathbf{S_D}))$:

$$R(p,A) \leftarrow R_D(p,A), c_R(A), \neg S_D(p,\_) \tag{6.56}$$
$$R(p,A) \leftarrow R_D(p,A), c_R(A), R_D(p,\_) \tag{6.57}$$
$$R(p,A) \leftarrow R_D(p,A), c_R(A), S_D(p,A'), \neg c_R(A') \tag{6.58}$$
$$R(p,A) \leftarrow R_D(p,A), R_D(p,A), \neg c_R(A) \tag{6.59}$$

$$R(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), c_R(A), \neg \mathbf{S_D}(\mathbf{p},\mathbf{A}) \tag{6.60}$$
$$R(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), c_R(A), \mathbf{R_D}(\mathbf{p},\_) \tag{6.61}$$
$$R(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), \mathbf{c_R}(\mathbf{A}), \neg \mathbf{c_R}(\mathbf{A}) \tag{6.62}$$
$$R(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), \mathbf{R_D}(\mathbf{p},\mathbf{A}), \neg c_R(A) \tag{6.63}$$

$$S(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), c_S(A), \neg \mathbf{R_D}(\mathbf{p},\mathbf{A}), \neg S_D(p,\_) \tag{6.64}$$
$$S(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), c_S(A), \neg \mathbf{R_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_) \tag{6.65}$$
$$S(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), c_S(A), \neg \mathbf{R_D}(\mathbf{p},\mathbf{A}), S_D(p,A'), R_D(p,A), A' = A \tag{6.66}$$
$$S(p,A) \leftarrow R_D(p,A), c_S(A), \mathbf{S_D}(\mathbf{p},\_), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.67}$$
$$S(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), c_S(A), S_D(p,\_), \neg \mathbf{R_D}(\mathbf{p},\_) \tag{6.68}$$
$$S(p,A) \leftarrow R_D(p,A), c_S(A), S_D(p,\_), S_D(p,A'), R_D(p,A), A' = A \tag{6.69}$$
$$S(p,A) \leftarrow R_D(p,A), \mathbf{c_S}(\mathbf{A}), \neg \mathbf{c_S}(\mathbf{A}), \neg S_D(p,\_) \tag{6.70}$$
$$S(p,A) \leftarrow R_D(p,A), \mathbf{c_S}(\mathbf{A}), \neg \mathbf{c_S}(\mathbf{A}), \neg R_D(p,A) \tag{6.71}$$
$$S(p,A) \leftarrow R_D(p,A), \mathbf{c_S}(\mathbf{A}), \neg \mathbf{c_S}(\mathbf{A}), S_D(p,A'), R_D(p,A), A' = A \tag{6.72}$$
$$S(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), c_S(A), \neg R_D(p,\_), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.73}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), c_S(A), \neg R_D(p,\_), \neg R_D(p,\_) \tag{6.74}$$
$$S(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), c_S(A), \neg R_D(p,\_), S_D(p,A), \mathbf{R_D}(\mathbf{p},\mathbf{A''}), A = A'' \tag{6.75}$$
$$S(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), c_S(A), S_D(p,\_), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.76}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), c_S(A), S_D(p,\_), \neg R_D(p,\_) \tag{6.77}$$
$$S(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), c_S(A), S_D(p,\_), S_D(p,A), \mathbf{R_D}(\mathbf{p},\mathbf{A''}), A = A'' \tag{6.78}$$
$$S(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), c_S(A), R_D(p,A'), c_S(A'), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.79}$$
$$S(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), c_S(A), \mathbf{R_D}(\mathbf{p},\mathbf{A'}), c_S(A'), \neg R_D(p,\_) \tag{6.80}$$
$$S(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), c_S(A), \mathbf{R_D}(\mathbf{p},\mathbf{A'}), c_S(A'), S_D(p,A), R_D(p,A''), A = A'' \tag{6.81}$$

$$S(p,A) \leftarrow S_D(p,A), R_D(p,A'), A \neq A' \tag{6.82}$$

$$S(p,A) \leftarrow R_D(p,A), \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg c_S(A), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.83}$$
$$S(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), S_D(p,A), \neg c_S(A), \neg \mathbf{R_D}(\mathbf{p},\_) \tag{6.84}$$
$$S(p,A) \leftarrow R_D(p,A), S_D(p,A), \neg c_S(A), S_D(p,A), R_D(p,A), A = A \tag{6.85}$$
$$S(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), S_D(p,A), \neg c_S(A), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.86}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), S_D(p,A), \neg c_S(A), \neg R_D(p,\_) \tag{6.87}$$
$$S(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), S_D(p,A), \neg c_S(A), S_D(p,A), \mathbf{R_D}(\mathbf{p},\mathbf{A'}), A = A') \tag{6.88}$$

$$T'(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), \neg c_R(A), \neg c_S(A), \neg \mathbf{R_D}(\mathbf{p},\_), \neg S_D(p,\_) \tag{6.89}$$
$$T'(p,A) \leftarrow \mathbf{R_D}(\mathbf{p},\mathbf{A}), \neg c_R(A), \neg c_S(A), \neg \mathbf{R_D}(\mathbf{p},\_), S_D(p,A'), c_S(A') \tag{6.90}$$
$$T'(p,A) \leftarrow R_D(p,A), \neg \mathbf{c_R}(\mathbf{A}), \neg c_S(A), R_D(p,A), \mathbf{c_R}(\mathbf{A}), \neg S_D(p,\_) \tag{6.91}$$
$$T'(p,A) \leftarrow R_D(p,A), \neg \mathbf{c_R}(\mathbf{A}), \neg c_S(A), R_D(p,A), \mathbf{c_R}(\mathbf{A}), S_D(p,A''), c_S(A'') \tag{6.92}$$

$$T'(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), \neg c_R(A), \neg c_S(A), \neg R_D(p,\_), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.93}$$
$$T'(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), \neg c_R(A), \neg \mathbf{c_S}(\mathbf{A}), \neg R_D(p,\_), S_D(p,A), \mathbf{c_S}(\mathbf{A})) \tag{6.94}$$
$$T'(p,A) \leftarrow \mathbf{S_D}(\mathbf{p},\mathbf{A}), \neg R_D(p,\_), \neg c_R(A), \neg c_S(A), R_D(p,A'), c_R(A'), \neg \mathbf{S_D}(\mathbf{p},\_) \tag{6.95}$$
$$T'(p,A) \leftarrow S_D(p,A), \neg \mathbf{R_D}(\mathbf{p},\_), \neg c_R(A), \neg c_S(A), \mathbf{R_D}(\mathbf{p},\mathbf{A}'), c_R(A'), S_D(p,A''), c_S(A'') \tag{6.96}$$

Using Lemma 4 we remove all rules that have contradicting literals (marked bold). Particularly, there remains no rule for T' as expected. Further, we remove duplicate literals within the rules according to Lemma 7, so we obtain the simplified rule set:

$$R(p,A) \leftarrow R_D(p,A), c_R(A), \neg S_D(p,\_) \tag{6.97}$$
$$R(p,A) \leftarrow R_D(p,A), c_R(A) \tag{6.98}$$
$$R(p,A) \leftarrow R_D(p,A), c_R(A), S_D(p,A'), \neg c_R(A') \tag{6.99}$$
$$R(p,A) \leftarrow R_D(p,A), \neg c_R(A) \tag{6.100}$$

$$S(p,A) \leftarrow S_D(p,A), R_D(p,A), c_S(A) \tag{6.101}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), c_S(A) \tag{6.102}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), c_S(A) \tag{6.103}$$
$$S(p,A) \leftarrow S_D(p,A), R_D(p,A'), A \neq A' \tag{6.104}$$
$$S(p,A) \leftarrow S_D(p,A), R_D(p,A), \neg c_S(A) \tag{6.105}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_), \neg c_S(A) \tag{6.106}$$

Rule 6.101 is derived from Rule 6.69 by applying the equivalence of A and A' to the remaining literals. Let's now focus on the rules for R. According to Lemma 8, Rules 6.97 and 6.99 are subsumed by Rule 6.98, since they contain the identical literals as Rule 6.98 plus additional conditions. Lemma 3 allows us to further reduce Rules 6.98 and 6.100, so we obtain that all tuples in R survive one round trip without any information loss or gain:

$$R(p,A) \leftarrow R_D(p,A) \tag{6.107}$$

We also reduce the rules for S. Rule 6.103 can be removed since it is equal to Rule 6.102. With Lemma 3, Rules 6.102 and 6.106 as well as Rules 6.101 and 6.105 can be combined respectively. This results in the following rules for S:

$$S(p,A) \leftarrow S_D(p,A), R_D(p,A) \tag{6.108}$$
$$S(p,A) \leftarrow S_D(p,A), \neg R_D(p,\_) \tag{6.109}$$
$$S(p,A) \leftarrow S_D(p,A), R_D(p,A'), A \neq A' \tag{6.110}$$

Rules 6.108 and 6.110 basically state that the payload data in R (A and A' respectively) is either equal to or different from the payload data in S for the same key p. When we rewrite Rule 6.108 to:

$$S(p,A) \leftarrow S_D(p,A), R_D(p,A'), A = A' \tag{6.111}$$

we can apply Lemma 3 to obtain the two rules:

$$S(p, A) \leftarrow S_D(p, A), R_D(p, \_) \tag{6.112}$$
$$S(p, A) \leftarrow S_D(p, A), \neg R_D(p, \_) \tag{6.113}$$

With the help of Lemma 3, we reduce $\gamma_{trg}(\gamma_{src}(R_D, S_D))$ to

$$R(p, A) \leftarrow R_D(p, A) \tag{6.114}$$
$$S(p, A) \leftarrow S_D(p, A) \quad \square \tag{6.115}$$

So, both Condition 6.2 and Condition 6.1 for the data independence of the PARTITION SMO are formally validated by now. Since the MERGE SMO is the inverse of the PARTITION SMO and uses the exact same mapping rules vice versa, we also implicitly validated the data independence of the MERGE SMO. This formal evaluation works for the remaining BiDEL SMOs, as well (Appendix A). BiDEL's SMOs ensure that given data at any schema version V_n that is propagated and stored at a direct predecessor V_{n-1} or direct successor schema version V_{n+1} can always be read completely and correctly in V_n. In the following, we will also show that the data independence conditions also hold for write operations (Section 6.4.3) as well as for chains of SMOs (Section 6.4.4).

6.4.3 Write operations

Data independence is also required for write operations: When updating a not-materialized schema version, this update is propagated to the materialized schema in a way that it is correctly reflected when reading the updated data again. Given a materialized SMO and data D_{src} at the source side, we apply a write operation $\Delta_{src}(D_{src})$ to D_{src}. Thereby, $\Delta_{src}(D_{src})$ can both insert and update and delete data. Initially, we store D_{src} at the target side using $D_{trg} = \gamma_{trg}(D_{src})$. To write at the source side, we have to temporarily map back the data to the source with $\gamma_{src}^{data}(D_{trg})$, apply the write Δ_{src}, and map the updated data back to target side with $D'_{trg} = \gamma_{trg}(\Delta_{src}(\gamma_{src}^{data}(D_{trg})))$. Reading the data from the updated target $\gamma_{src}^{data}(D'_{trg})$ has to be equal to applying the write operation $\Delta_{src}(D_{src})$ directly on the source side.

$$\Delta_{src}(D_{src}) = \gamma_{src}^{data}(\gamma_{trg}(\Delta_{src}(\gamma_{src}^{data}(\gamma_{trg}(D_{src})))))) \tag{6.116}$$

We have already shown that $D = \gamma_{src}^{data}(\gamma_{trg}(D))$ holds for any data D at the target side, so that Equation 6.116 reduces to $\Delta_{src}(D_{src}) = \Delta_{src}(D_{src})$. Hence writes are correctly propagated through the SMOs. The same holds vice versa for writing at the target-side of virtualized SMOs:

$$\Delta_{trg}(D_{trg}) = \gamma_{trg}^{data}(\gamma_{src}(\Delta_{trg}(\gamma_{trg}^{data}(\gamma_{src}(D_{trg})))))) \tag{6.117}$$

6.4.4 Chains of SMOs

The data independence of BiDEL SMOs also holds for sequences of SMOs: $S = (smo_1, \ldots smo_n)$, where $\gamma_{i,src/trg}$ is the respective mapping of smo_i and the source database D_{src} is evolved to the target database D_{trg}: $D_{src} \xrightarrow{S} D_{trg}$. Analogous to symmetric relational lenses [65], there are no side-effects between multiple BiDEL SMOs. Thus, BiDEL's data independence is also guaranteed along sequences of SMOs:

$$D_{src} = \gamma_{1,src}^{data}(\ldots \gamma_{n,src}(\gamma_{n,trg}(\ldots \gamma_{1,trg}(D_{src})))) \tag{6.118}$$
$$D_{trg} = \gamma_{n,trg}^{data}(\ldots \gamma_{1,trg}(\gamma_{1,src}(\ldots \gamma_{n,src}(D_{trg})))) \tag{6.119}$$

This ensures data independence, since any schema version can now be read and written without information loss or gain, no matter in which table versions the data is actually materialized. The auxiliary tables keep the otherwise lost information and we have formally validated their feasibility. With the formal guarantee of data independence—also along sequences of SMOs and for write operations—we have laid a solid formal foundation for InVerDa's delta code generation.

In sum, we can now safely assume data independence for BiDEL-based evolutions: Wherever we materialize the data physically, it is guaranteed that each table version can be accessed just like a regular table—no data will be lost or gained. To our best knowledge, we are the first to design a set of powerful SMOs and validate their data independence according to the criteria of symmetric relational lenses. This is a strong guarantee and the basis for InVerDa, as we can now guarantee that any schema version in a multi-schema-version database system will behave like a regular single-schema database, which allows developers to develop their applications just as usual but quickly evolve the schema of the underlying database when needed.

6.5 EMPIRICAL EVALUATION

InVerDa's co-existing schema versions bring great advantages for software systems by decoupling the different goals of different stakeholders. Users can continuously access all schema versions, while developers can focus on the actual continuous implementation of the software without caring about former versions. Above all, the DBA can change the physical table schema of the database to optimize the overall performance without restricting the availability of the co-existing schema versions or invalidating the developers' code. We have already formally evaluated the correctness of InVerDa's support for co-existing schema versions, so developers and DBAs can safely use the provided features without risking inconsistencies or information loss in the stored data. Now, we further evaluate our prototypical implementation empirically in more detail to show our success and further potentials. InVerDa automatically generates the delta code based on the discussed Datalog rules; in Section 6.5.1, we measure the overhead of accessing data through InVerDa's delta code compared to a handwritten SQL implementation of co-existing schema versions and show that it is reasonably small. In Section 6.5.2, we show that the possibility to easily adapt the physical table schema to a changed workload with the click of a button outweighs the small overhead of InVerDa's automatically generated delta code. Materializing the data according to the most accessed version speeds up the data access significantly.

Setup: For the measurements, we use two different data sets to gather a holistic idea of InVerDa's characteristics. We use (1) our TasKy example as a small-sized but comprehensive scenario and, (2) 171 schema versions of Wikimedia [40] as a long real-world scenario. We measure single thread performance of a PostgreSQL 9.4 database with co-existing schema versions on a Core i7 machine with 2,4GHz and 8GB memory.

6.5.1 Overhead of Generated Delta Code

InVerDa's delta code is generated from Datalog rules and aims at a general and solid solution. So far, our focus is on the correct propagation of data access on multiple co-existing schema versions. We expect the database optimizer to find a fast execution plan, which is an optimistic assumption in practice. There will be an overhead of InVerDa compared to hand-optimized SQL but we will show that it is reasonably small, which we attribute to the simple access pattern InVerDa generates from the Datalog rules, so the optimizer is not overly challenged with complex and tricky queries.

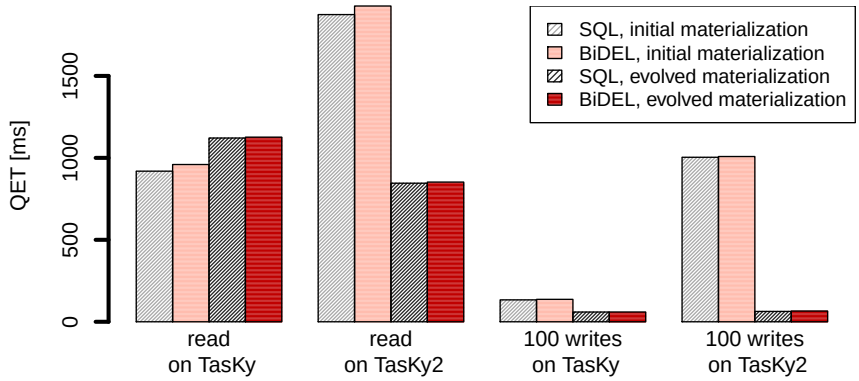
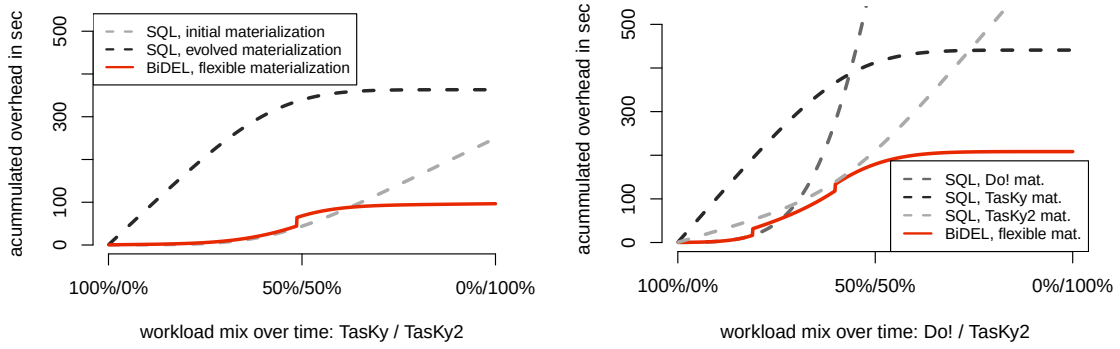


Figure 6.5: Overhead of generated code.

TasKy: As already used in the empiric evaluation of BiDEL's bidirectionality in Section 4.3.3, we implement the evolution from **TasKy** to **TasKy2** with handwritten and hand-optimized SQL to compare its performance to the semantically equivalent evolution with BiDEL. Again, we assume a multi-schema-version database system, where both **TasKy** and **TasKy2** co-exist and the physical materialization will be migrated from **TasKy** to **TasKy2** eventually. Therefore, we implemented (1) creating the initial **TasKy** schema, (2) creating the additional schema version **TasKy2** with the respective views and triggers, and (3) migrating the physical table schema to **TasKy2** and adapting all delta code.

The automated delta code generation does not only eliminate the error-prone and expensive such a manual implementation, but it is also reasonably fast. Creating the initial schema version of the **TasKy** example (Figure 2.4, page 28) and making it available as a new schema version for applications took $154\,\mathrm{ms}$ on our test system. The evolution to **TasKy2**, which includes two SMOs, requires $230\,\mathrm{ms}$ for both the generation and execution of the evolution script. The same took $177\,\mathrm{ms}$ for **Do!**. Doing the same with handwritten SQL was equally fast; any differences lay within the measuring inaccuracy. Please note that the complexity of generating and executing evolution scripts depends linearly on the number of SMOs N and the number of untouched table versions M. The complexity is $O(N + M)$, since we generate the delta code for each SMO locally and exclusively work on the neighboring table versions. This principle protects against additional complexity in longer sequences of SMOs. The same holds for the complexity of executing migration scripts. It is $O(N)$ since InVerDa merely moves the data and updates the delta code for the respective SMOs stepwise.

In Figure 6.5, we use the **TasKy** and thy **TasKy2** schema versions with $100\,000$ tasks and compare the performance of the InVerDa generated delta code to the handwritten one. The charts show the query execution time (1) for reading the **TasKy** schema, (2) for reading the **TasKy2** schema, (3) for inserting a tuple to **TasKy**, and (4) for inserting a tuple to **TasKy2**. We evaluate both the initial materialization according to **TasKy** and the evolved materialization according to **TasKy2**. There are two aspects to observe. First, the hand-optimized delta code causes slightly less (up to $4\,\%$) overhead than the generated one. Keeping in mind the difference in length and complexity of the code ($359\,\mathrm{x}$ LOC for the evolution), a performance overhead of $4\,\%$ in average is more than reasonable for most users. Second, the materialization significantly influences the actual performance. Reading the data in the materialized version is up to twice as fast as accessing it from the respective other version in this scenario. For the write workload (insert new tasks), we observe again a reasonably small overhead compared to handwritten SQL. Interestingly, the evolved materialization is always faster because the initial materialization requires to manage an additional auxiliary table for the foreign key relationship. A DBA can optimize the overall performance for a given workload by adapting the materialization, which is itself a very simple task with InVerDa. Since the solution space of possible materializations can grow very large for long evolution histories, InVerDa also comes with an adviser that takes the current workload and proposes a feasible materialization with a high overall performance. The adviser will be presented in Chapter 7.



(a) Flexible materialization for TasKy and TasKy2. (b) Flexible materialization for Do!, TasKy, and TasKy2.

Figure 6.6: Benefits of flexible materialization with InVerDa.

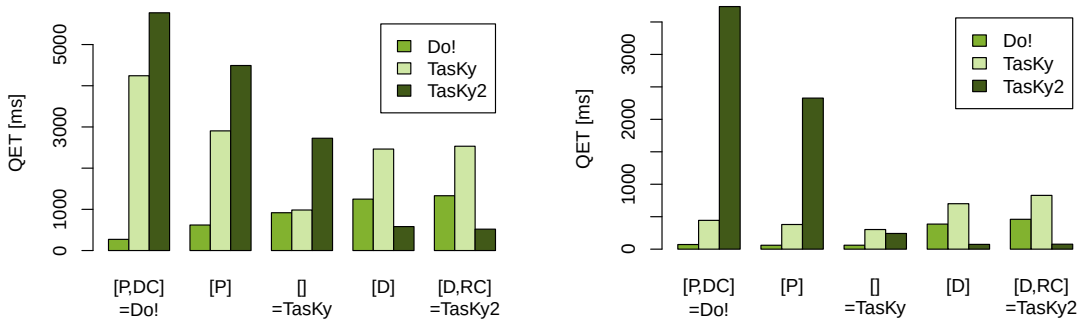
6.5.2 Benefit of Flexible Materialization

Although InVerDa introduces a slight overhead when accessing data, it provides a great benefit by allowing for easy adaptation of the physical table schema to the current workload. Adapting the physical table schema to the current workload is hard with handwritten SQL, but almost for free with InVerDa (1 LOC instead of 182 in our TasKy example). Let's assume a development team spares the effort for rewriting delta code and works with a fixed materialization. Then, InVerDa allows to easily increase the overall performance, as we will show using the TasKy example and the Wikimedia scenario.

TasKy (macro benchmark): Again, we use the TasKy example with 100 000 tasks. Figure 6.6(a) shows the accumulated propagation overhead for handwritten SQL with the two fixed materialization and for InVerDa with an adaptive materialization. Assume, over time the workload changes from 0 % access to TasKy2 and 100 % to TasKy to the opposite 100 % and 0 % according to the Technology Adoption Life Cycle. The adoption is divided into 1000 time slices where 1000 queries are executed respectively. The workload mixes 50 % reads, 20 % inserts, 20 % updates, and 10 % deletes. As soon as the evolved materialization is faster for the current workload mix, we instruct InVerDa to change the materialization. As can be seen, InVerDa facilitates significantly better performance—including migration cost—than a fixed materialization.

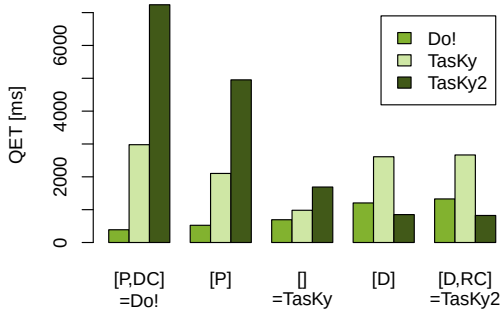
This effect increases with the length of the evolution since InVerDa can also materialize intermediate stages of the evolution history. Assume, all users use exclusively the mobile phone app Do!; but as TasKy2 gets released users switch to TasKy2 which comes with its own mobile app. In Figure 6.6(b), we simulate the accumulated overhead for either materializing one of the three schema versions or for a flexible materialization. InVerDa's flexible materialization naturally starts at Do!, but moves to TasKy after several users started using TasKy2, and finally moves to TasKy2 when the majority of users also did so. Again, InVerDa's flexible materialization significantly reduces the overhead for data propagation without any interaction of any developer.

TasKy (micro benchmark): The DBA can choose between multiple materialization schemas. InVerDa generates the delta code to move the data and also adapt existing delta code to keep all coexisting schema versions alive. The number of valid materialization schemas greatly depends on the actual structure of the evolution. The lower bound is a linear sequence of depending SMOs, e.g. one table with N `ADD COLUMN` SMOs has N valid materializations. The upper bound are N independent SMOs, each evolving another table, with 2^N valid materializations. Specifically, the TasKy example has five valid materializations.



(a) Materializations for TasKy read.

(b) Materializations for TasKy write.



(c) Materializations for TasKy mix.

Figure 6.7: Different workloads on all possible materialization of TasKy.

Figure 6.7 shows the data access performance on the three schema versions for each of the five materialization schema. The materialization schemas are represented as the lists of SMOs that are materialized. We use abbreviations of the SMOs: e.g. $[D, RC]$ on the very right corresponds to schema version TasKy2 since both the DECOMPOSE (D) and the RENAME COLUMN (RC) SMO are materialized. The initial materialization is in the middle, while the materialization according to Do! is on the very left. The workload mixes 50 % reads, 20 % inserts, 20 % updates, 10 % deletes in Figure 6.7(c), 100 % reads in Figure 6.7(a), and 100 % inserts in Figure 6.7(b) on the depicted schema versions. Again, the measurements show that accesses to each schema version are fastest when its respective table versions are materialized, i.e. when the physical table schema fits the accessed schema version. However, there are differences in the actual overhead, so the globally optimal materialization depends on the workload distribution among the schema version. E.g. writing to TasKy2 is 49 times faster when the physical table schema matches TasKy2 instead of Do!. This gain increases with every SMO, so for longer evolutions with more SMOs it will be even higher.

Wikmedia: The benefits of the flexible materialization originate from the increased performance when accessing data locally without the propagation through SMOs. We load our Wikimedia with the data of Akan Wiki in schema version *v16524* (109th version) that contains 14 359 pages and 536 283 links. We measure the read performance for the template queries from [40] both in schema version *v04619* (28th version) and *v25635* (171th version). The chosen materializations match version *v01284* (1st), *v16524* (109th), and *v25635* (171th) respectively. In Figure 6.8, a great performance difference of up to two orders of magnitude is visible, so there is a huge optimization potential. We attribute the asymmetry to the dominance of ADD COLUMN SMOs, which need an expensive join with an auxiliary table to propagate data forwards, but only a comparable cheap projection to propagate backwards.

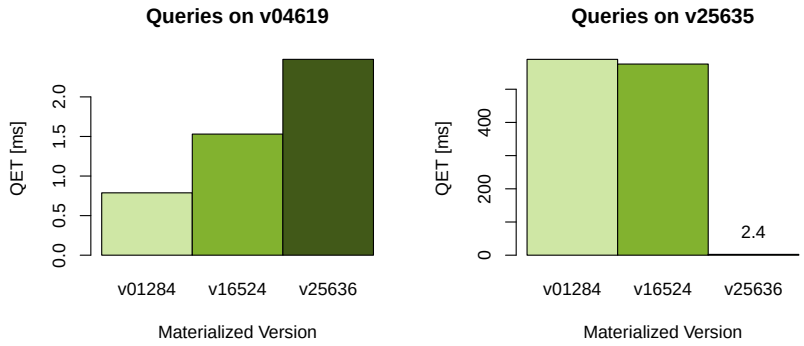


Figure 6.8: Optimization potential for Wikimedia.

In sum, the evaluation has shown that the performance overhead of the InVerDa-generated delta code in comparison to hand-written delta code is very small (4 %) and a reasonable tradeoff for the saved implementation effort in most scenarios. Further, the data independence of the co-existing schema versions provides a huge optimization potential: Changing the materialization so that applications do access materialized table versions mostly directly, increases the performance by orders of magnitude. Since InVerDa allows to change the materialization with the click of button instead of error-prone and expensive manual migrations, we can safely adapt the materialization to always match the workload as good as possible. Hence, the data independence provided by InVerDa allows using the optimization potential easily and without any risk of losing data during the migration.

6.6 SUMMARY

In this chapter, we introduced the materialization-independent mapping semantics of BiDEL, which especially includes the management of auxiliary information for the non-redundant or partially redundant materializations in multi-schema-version database systems. The major objective was that the SMOs can propagate data that is written at any non-materialized table version to the physically materialized table versions without any information loss—since not all SMOs are information preserving, the other-wise lost information needs to be managed in the auxiliary tables. When reading again in the non-materialized table version, we expect to see exactly the initially written data again without any information being lost or gained.

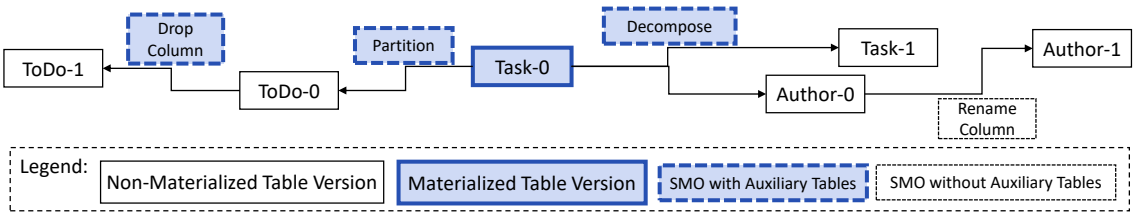
Representing the materialization-independent mapping semantics of BiDEL's SMOs with sets of Datalog rules allowed us to immediately generate delta code for propagating or migrating data through SMOs to realize truly co-existing schema versions that can all be accessed at the same time. We formally evaluated the data independence, so we can now guarantee that no matter which table versions we materialize, from the applications' perspective each schema version behaves like a common single-schema database without any information loss or gain. We have done this formal evaluation for non-redundant materializations, since this is the hardest. We can apply the results to partial or fully redundant materializations as well. Starting with a non-redundant materialization, it is straight forward to materialize additional table versions: Instead of reconstructing the table version with the help of auxiliary tables, we directly access the now materialized table—the remainder of the data access propagation remains unchanged.

Further, we empirically analyzed the benefits of the independent materialization, which allows to increase the overall performance significantly by basically moving the materialization according to the current workload with the click of a button. Just like the whole chapter, we focused the evaluation on non-redundant materializations. In the next chapter, we equip DBAs with an adviser that automatically proposes the best set of table versions to materialize for the current workload, which then covers partially or fully redundant materializations as well.

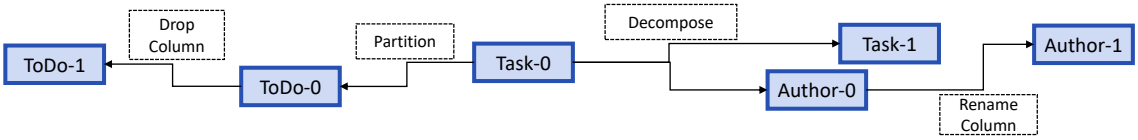
ADVISER FOR MATERIALIZING CO-EXISTING SCHEMA VERSIONS

Co-existing schema versions within a multi-schema-version database system are different representations of the same conceptual data set. Thanks to the bidirectionality of BiDEL's SMOs, there is no need to materialize each and every schema version to let them co-exist, but we can also non-redundantly store the data in one slice of the schema evolution history. The two extremes of fully redundant and non-redundant materialization span a huge search space of partially redundant materializations, which is worth exploring to gain significant speedups as shown before in Section 6.4. However, for a DBA it is very hard to find the best or at least a good materialization manually—especially when the workload continuously changes and the database is further evolved to new schema versions, finding the best materialization in the huge search space is a never ending fight.

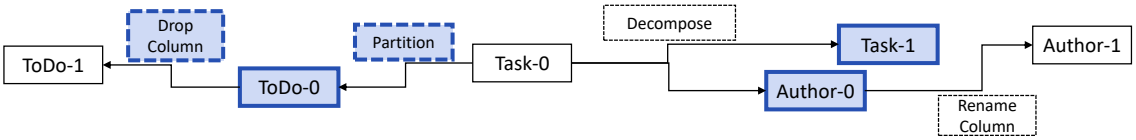
In this chapter, we present an adviser for InVerDa that takes the current workload and the current schema versions catalog and proposes an optimized physical materialization, which yields significant performance improvements. Whenever the workload changes or whenever developers create or drop schema versions, the DBA easily calls the adviser again and adapts the database's materialization to the new situation. At first, we discuss the objectives for such an adviser in Section 7.1 and detail on the optimization problem in Section 7.2. Since, we cannot physically create every single materialization to test its performance, we aim for a cost model-based optimization. In Section 7.3, we conduct a detailed analysis of the characteristics of the overhead for propagating any read or write accesses through InVerDa's delta code and derive a stable cost model for SMO-based evolutions. In Section 7.4, we define the space of all possible materializations for a given schema versions catalog and present an evolutionary optimization algorithm in Section 7.5 that explores the search space to find the best materialization based on the discussed cost function. Finally, we evaluate both the cost model and the optimizer in Section 7.6 and summarize the chapter in Section 7.7.



(a) Non-redundant materialization in preferred table versions.



(b) Fully redundant materialization in all table versions.



(c) Partially redundant materialization as tradeoff between read performance, write performance, and space constraints.

Figure 7.1: Possible materializations for our TasKy example from Figure 2.9 on page 37.

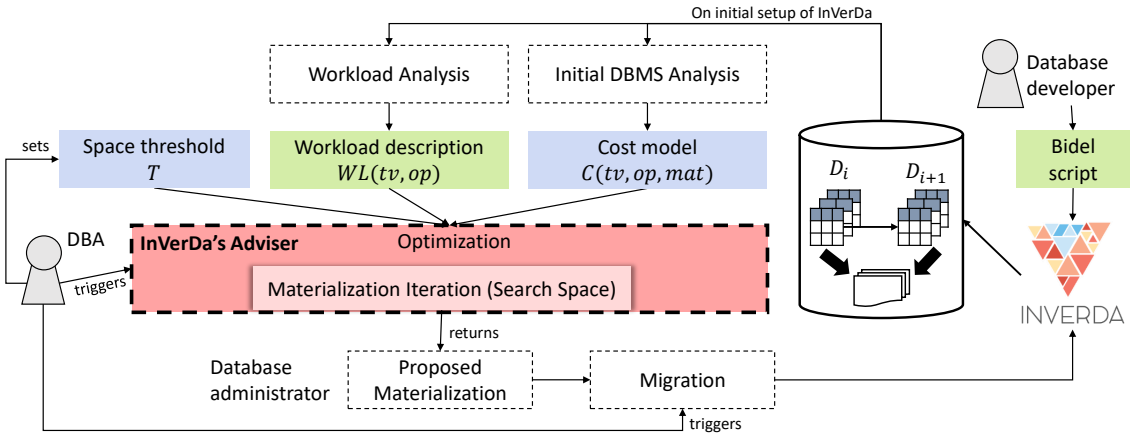


Figure 7.2: Workflow of InVerDa's adviser.

7.1 OBJECTIVES

We will use the freedom of full data independence in multi-schema-version database systems to materialize a set of table versions that provide the best performance for the current workload. By bringing the data as close as possible to the actually accessed schema versions, the propagation overhead for both reading and writing is reduced. When the workload is spread over multiple schema versions, we can also replicate the data to multiple table versions in the schema versions catalog to obtain short access paths for reading with a reasonable amount of materialized table versions—however, with each table version we materialize additionally, the space requirements as well as the overhead for write operations increases. Figure 7.1 shows three different possible extents of redundancy for our TasKy example (Figure 2.9, page 37). We can either have no, full, or partial redundancy. In the non-redundant materialization (Figure 7.1(a)), only one preferred schema version is materialized; accesses to all other versions are propagated to this preferred version and to the auxiliary tables, e.g. the initial TasKy version is preferred in the depicted example. In contrast, the fully redundant materialization (Figure 7.1(b)), physically stores every single table version, which requires way more storage space, however, no auxiliary tables are needed, which speeds up query processing. Finally, the partially redundant materialization (Figure 7.1(c)) strikes a balance between the two extremes, by materializing a subset of the table versions that is cost-wise as close as possible to the accessed table versions.

Interestingly, the overhead for writing to additional materialized table versions is not as large as one would expect: Since every SMO can manage auxiliary tables even if the neighboring table versions are not materialized, write operations need to be propagated logically through all SMOs either way to compute the updates on the auxiliary tables as well. The overhead for writing to materialized table versions is then only the physical write operation to the table version but not the calculation of the propagated write operation itself. Hence, the penalty for additionally writing the payload data to a materialized table version might be a reasonable price to pay—we examine this in detail. The resulting optimization problem that needs to be solved by the adviser is to find the best set of table versions to materialize, which is a tradeoff between read performance, write performance, and given space requirements.

An adviser should propose an optimized materialization for the co-existing schema versions in a multi-schema-version database system based on the workload model for all schema versions, the cost model for propagating data accesses between schema versions, and an upper bound for the available space that controls the maximum degree of redundancy. Thereby, the envisioned adviser should roughly follow a workflow as shown in Figure 7.2. The developers use BiDEL evolution scripts to let InVerDa

create co-existing schema versions within the database. When the DBA triggers InVerDa's adviser, it should determine the best materialization for the current workload under a given **space threshold** based on a cost model that is individually learned for the database. The adviser should be **cost model**-based. It is not applicable to temporarily change the materialization to exactly measure the costs, because the migration to one possible materialization can easily take several minutes to multiple days in realistic scenarios. Instead, an adviser should use a cost model to quickly estimate the costs. Further, InVerDa's adviser should analyze the workload among all versions in the multi-schema-version database system and represent it in a **workload model** which contains the percentage of SQL operations for each individual table version. Given the workload model, the upper space bound, and the cost model, the optimizer should determine the set of materialized table versions that provides the best overall performance. Finally, the DBA should be able to confirm the proposed materialization and instruct InVerDa to migrate the database accordingly and automatically generate new delta code to keep all table versions in all schema versions alive—the table versions now access the newly materialized table versions with an increased overall performance. In sum, we expect the adviser to propose a set of table versions that should be materialized in order to obtain the best performance for the current workload on the given system with given space boundaries.

7.2 PROBLEM DESCRIPTION

In the following, we discuss all the artifacts shown in Figure 7.2 in more detail to precisely specify the actual optimization problem—the objective of InVerDa's adviser. To do so, we introduce a formal notation:

- $\mathcal{SV}$ is the set of all external schema version in the schema versions catalog.
- $\mathcal{TV}(s)$ is the set of all table versions within the schema version $s \in \mathcal{SV}$.
- $\mathcal{TV}$ is the set of all table versions, which particularly includes those table versions that are merely intermediate steps between schema versions, $\{t | t \in \mathcal{TV}(s), s \in \mathcal{SV}\} \subseteq \mathcal{TV}$.
- $\mathcal{O} = \{\text{select, insert, update, delete}\}$ is the set of operations that can be executed on any table version $tv \in \mathcal{TV}(s)$ for all $s \in \mathcal{SV}$.
- $\mathcal{MATS} \subseteq \{\mathcal{M} | \mathcal{M} \subseteq \mathcal{TV}\}$ is the set of all valid materialization states, where $\mathcal{M}$ is a set of table versions that is physically materialized in that respective materialization.
- $\mathcal{S}(\mathcal{M})$ is the size of the given materialization $\mathcal{M} \in \mathcal{MATS}$.
- $\mathcal{T}$ is the specified threshold for the maximum size of the materialization.
- $\mathcal{E}(c)$ is the set of all directed hyper edges (SMOs) that are connected to the given table version $c \in \mathcal{TV}$.
- $\mathcal{N}(e, c)$ is the set of all table versions that are at the opposite end from the directed hyper edge (SMO) than the given table version $c \in \mathcal{TV}$; e.g. given a source table $c \in \mathcal{TV}$ of an SMO e then $\mathcal{N}(e, c)$ return all target table versions of the SMO e.

table version	select	insert	update	delete	sum
`Todo-1`	50 %	1 %	0 %	9 %	60 %
`Task-0`	5 %	2 %	2 %	1 %	10 %
`Task-1`	3 %	15 %	7 %	0 %	25 %
`Author-1`	2 %	2 %	1 %	0 %	5 %
sum	60 %	20 %	10 %	10 %	100 %

Table 7.1: Exemplary workload description for TasKy.

Workload description: Since the adviser works on the level of table versions, the workload itself is also given on the level of table versions. Obviously, if we have a workload given on the schema version level we can approximately transform this into a workload on table versions. To query the workload definition, we define a function $\mathcal{WL}(c, op)$, which returns the percentage of a given operation $op \in \mathcal{O}$ on a given table version $c \in \mathcal{TV}$—the function is only defined for table versions that are actually included in at least one schema version, since the other table versions are never explicitly accessed either way. The workload function's signature is:

$$\mathcal{WL}(c, op) : \{c | v \in \mathcal{SV}, c \in \mathcal{TV}(v)\} \times \mathcal{O} \rightarrow [0; 1] \tag{7.1}$$

For each table version, we define the global percentage of select, insert, update, and delete accesses. These percentages sum up to 100% over all table versions and all operations. Table 7.1 depicts an example workload distribution for our TasKy example (Figure 2.9, page 37). We assume that users mainly use the mobile Do! application to read their most important tasks and remove them after completion. While the TasKy versions is only used by a very small fraction of the users, there are many users that use the TasKy2 desktop application to create and update their tasks. Please note that `Todo-0` and `Author-0` are never accessed explicitly since they are in no external schema version.

Cost model: Propagating read and write accesses through SMOs naturally causes costs that need to be minimized by InVerDa's adviser. We define the cost model for the data propagation with the function $\mathcal{C}(c, \mathcal{M}, op)$, which takes a set of materialized table versions $\mathcal{M} \in \mathcal{MATS}$ and returns the costs for performing the given operation $op \in \mathcal{O}$ on the given table version $c \in \mathcal{TV}$:

$$\mathcal{C}(c, \mathcal{M}, op) : \mathcal{TV} \times \mathcal{MATS} \times \mathcal{O} \rightarrow \mathbb{R}^+ \tag{7.2}$$

Since the actual costs for propagating a read or a write operation through an SMO significantly depend on the underlying hardware and the used DBMS, the cost function will be learned automatically when initially setting up InVerDa. We will analyze the general characteristics of the cost function and propose a feasible learning algorithm in Section 7.3.

Optimization objective: Putting everything together, the optimization objective of InVerDa's adviser is to find the best set of materialized table versions $\mathcal{M} \in \mathcal{MATS}$ that is within the specified space threshold $\mathcal{T}$ and provides the minimal costs for accessing the data. The costs are calculated as the sum of all operations op on all table versions c weighted with their share of the overall workload as given by the workload $\mathcal{WL}(c, op)$. Hence, the overall optimization goal is to find the materialization $\mathcal{M}_{best} \in \mathcal{MATS}$ with:

$$\mathcal{M}_{best} = \underset{\mathcal{M} \in \mathcal{MATS}}{arg\,min} \left(\sum_{c \in \{c | v \in \mathcal{SV}, c \in \mathcal{TV}(v)\}} \left(\sum_{op \in \mathcal{O}} \mathcal{WL}(c, op) \cdot \mathcal{C}(c, \mathcal{M}, op) \right) \right) \wedge \mathcal{S}(\mathcal{M}) \leq \mathcal{T} \tag{7.3}$$

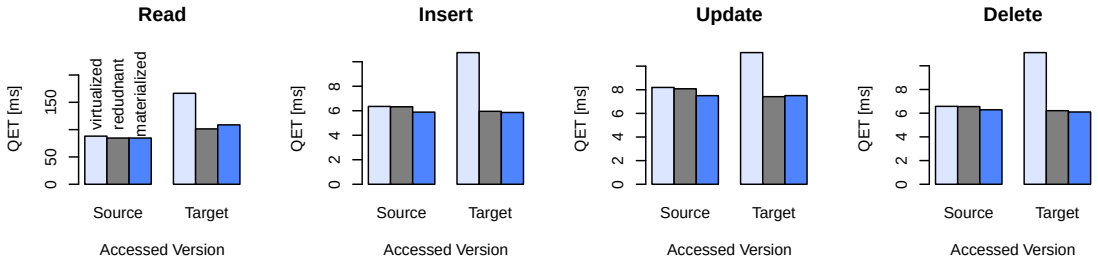


Figure 7.3: Accessing source/target table versions of `ADD COLUMN` SMO with different materializations.

Intuitively speaking, we minimize the expected query execution time for the given workload. In sum, we defined the workflow the adviser should follow and formalized all artifacts such as the given workload, the cost model, etc. to finally specify the optimization problem that should be solved by the adviser. In the following, we detail on the cost model $\mathcal{C}(c, \mathcal{M}, op)$ in Section 7.3, the search space $\mathcal{MATS}$ in Section 7.4 and the actual optimization algorithm to achieve this optimization objective in Section 7.5 before we evaluate the results in Section 7.6.

7.3 COST MODEL

To estimate the costs occurring for data access propagation in multi-schema-version database systems, we first have to understand the characteristics of the overhead. Therefore, we analyze a micro benchmark for the access propagation through single SMOs in Section 7.3.1 and through short chains of SMOs in Section 7.3.2. After this quantitative analysis of the data access propagation, we conduct a qualitative analysis of the data access propagation patterns in Section 7.3.3. Different access patterns entail different cost computations, hence understanding the access patterns is essential for defining the cost model. We focus on the analysis of the general characteristics. Since the specific costs largely depend on the delta code generation, the used DBMS, the used hardware, and so on, we learn the specific cost function for each system individually. In Section 7.3.4, we finally use the gained insights to define a general cost model that can be trained for a specific multi-schema-version database system.

7.3.1 Performance Overhead of single SMOs

Most SMOs have an **asymmetric overhead** w.r.t. the materialization; this means that the overhead for the propagation in one direction is smaller than in the other direction. For the adviser, it is essential to know these characteristics. We examine each Operation $op \in \mathcal{O}$ for each single SMO. For all such pairs of SMO and operation there are two dimensions that span a total of six different scenarios to analyze. First, the workload can access either the source or the target table versions. Second, we materialize either the source table version (virtualized SMO), or both the source and the target table version (redundant SMO), or only the target table version (materialized SMO). For each of the resulting six scenarios, we individually analyze the overhead for any data access operation.

Figure 7.3 shows the query execution for reading, inserting, updating, and deleting data in a very simple scenario: A table with 2 columns and 100 000 tuples is evolved to a new table version with an additional column. The experiments have been conducted on a PostgreSQL 9.4 database on a Core i7 machine with 2.4GHz and 8GB memory. As can be seen in the figure, the data access times in the discussed six scenarios are more or less constant unless the SMO is virtualized and we access the

evolved table version. In this case the data is primarily stored without the added column but accessed with this column; the ADD COLUMN SMO uses an auxiliary table to persist the data of this added column. As a consequence, reading data at the target side results in a join between the source table version and the auxiliary table. In all other cases, data is either accessed locally or the added column is merely projected away, which causes almost no overhead. Particularly, the costs for accessing data locally is identical no matter whether the respective other side of the SMO is materialized as well or not. Due to this observation, we boil down the six different scenarios to four scenarios along the two dimensions: (1) access either at target or at source side and (2) access either locally or remotely with auxiliary tables. For writing, we see the same pattern: InVerDa's delta code calculates the effective write operation for both table versions since writes need to be propagated along the schema version history either way—the additional overhead for actually persisting the propagated write operation is negligibly small compared to the involved trigger calls etc. Merely the calculation of the respective write operation on the auxiliary table causes an overhead as can be seen in Figure 7.3. As a result the write performance has similar characteristics as the read performance. The asymmetry of the ADD COLUMN SMO suggest to always materialize or replicate it, as long as space constraints and neighboring SMOs do not contradict this.

Without loss of generality, we use the ADD COLUMN SMO for explanation, because it is the most common one; the remaining SMOs behave similarly, however, the tendency of virtualizing or materializing is not always that clear, because it depends on the set of managed auxiliary tables. The respective measurements are listed in Appendix B. As expected, the DROP COLUMN SMO shows the exactly opposite behavior of ADD COLUMN, hence there is only a significant overhead when accessing the source version of the materialized DROP COLUMN SMO. The RENAME COLUMN SMO never causes a relevant overhead at all since it is realized with cheap projections. Both the JOIN and the DECOMPOSE SMO exist in multiple variants that all cause different performance characteristics. E.g. writing to the source side of a materialized DECOMPOSE SMO that creates a new foreign key causes a very high overhead, since new foreign keys are generated while writing tuples to the second output table—subsequently the newly generated foreign keys need to be written to the first output table as well. On the contrary, when writing to either source or target side of an inner JOIN on an arbitrary condition, the materialization has almost no impact on the performance, since the most overhead is caused by calculating the write operation on the data tables while updating the auxiliary tables comes almost for free. Further, there is an overhead for accessing the target of a virtualized MEGRE SMO, however, the overhead for accessing the source of a materialized MERGE is even higher. Again, the PARTITION behaves contrary. The asymmetries are not intuitively obvious for DBAs, as DBAs are not necessarily aware of the managed auxiliary tables that significantly determine the actual overhead. A cost model-based adviser that hides all the complexity, releases DBAs from the need to understand the asymmetries in the first place. Using the discovered asymmetries, we can tell for each SMO whether it is supposedly beneficial to either materialize it or virtualize it or whether this does not matter at all (like e.g. for renaming a table/column). This knowledge is a main ingredient for InVerDa's adviser.

For estimating the costs of the propagation through single SMOs, we further analyze their **scaling behavior** with respect to the size of the table versions. Figure 7.4 shows again the ADD COLUMN SMO with an increasing number of tuples. The measurements confirm that the already discovered asymmetries hold independently of the table size and that we can reduce the scenario to four cases of local/remote access on the source/target side. We further see that the query execution time increases linearly with the number of tuples in the table versions for read operations and stays almost constant for write operations. As a consequence, we can use functions of the form $ax + b$ with x being the source tables' size to describe the scaling behavior of access propagation through SMOs. For writing through an ADD COLUMN SMO, a constant function would be enough for the ADD COLUMN SMO, however, this does not apply to all SMOs, which motivates using a linear functions for writing as well. For instance, propagating an insert operation through a JOIN SMO requires to scan the whole other table for potential join partners—hence, the costs grow with the size of the existing tables. Our measurements did not indicate the need for higher order approximations, though, the presented concepts are not limited to $ax + b$ and easily support the extension to higher order polynomials as cost functions if needed.

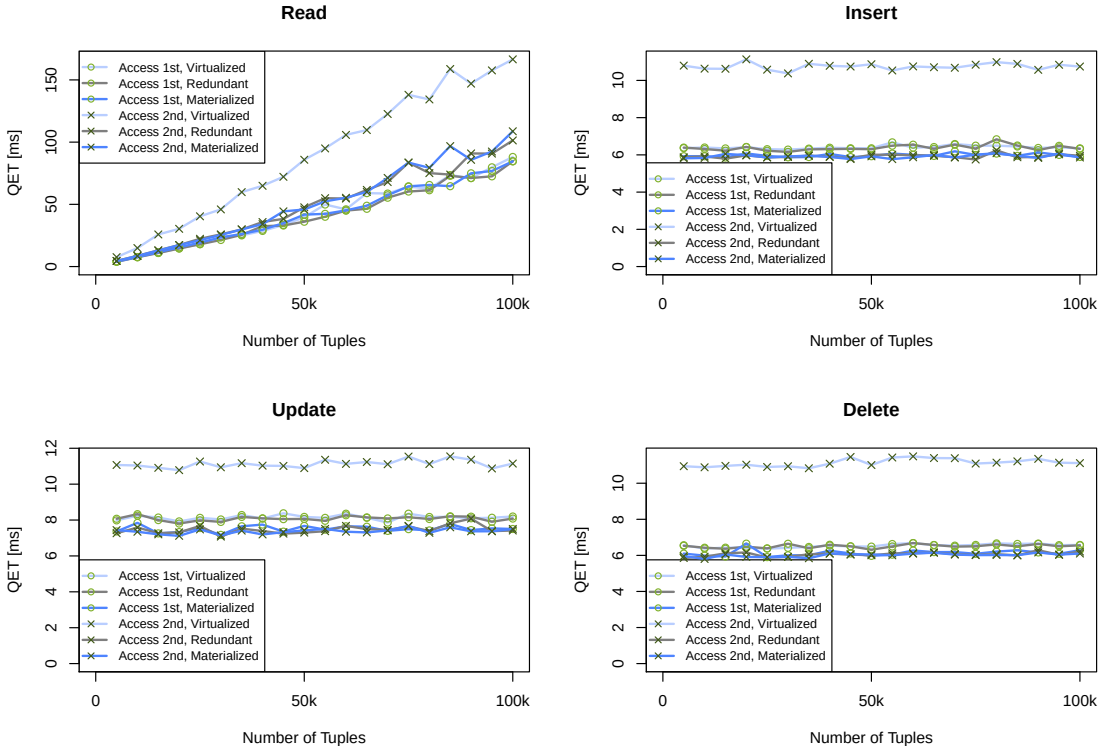


Figure 7.4: Scaling behavior for accesses propagation through an `ADD COLUMN` SMO.

7.3.2 Performance Overhead of Sequences of SMOs

To estimate the overhead for propagating data accesses through sequences of SMOs in a whole evolution, we now consider short evolutions to analyze how the overheads of the single SMOs combine. In fact, the overheads **combine linearly**, which confirms that it is always possible to gain a better performance by reducing the distances between data access and the materialized data. We conduct a micro benchmark on all possible evolutions with two subsequent SMOs—except of creating, dropping, and renaming tables since they do not have a performance overhead in the first place. We generate evolutions with two SMOs and three schema versions: first version v_1 – first SMO s_1 – second version v_2 – second SMO s_2 – third version v_3. The second version v_2 always contains a table $R(a, b, c)$; the number of generated tuples in this table is varied in the experiments. In the following, we analyze the improvement potential of the performance and check whether SMOs do positively or negatively impact each other or whether their overheads merely combine linearly. We analyze the performance of propagating both read and write operations through two subsequent SMOs.

Reading: In Figure 7.5, we exemplarily consider all the combinations of any SMO s_1 with an `ADD COLUMN` as second SMO s_2. Again, accessing data locally is up to twice as fast as propagating it through the SMOs, so the optimization potential exists in all scenarios. The expected performance for the sequential combination of both SMOs is calculated as the sum of propagating the data access through each SMO individually minus reading data locally at the second schema version. This is reasonable, because the source data for the second SMO is already in memory after executing the first one. Figure 7.5 shows that the measured time for propagating the data through two SMOs is always in the same range as the calculated **linear combination** of the two SMOs, so we show that there is great **optimization potential** for all combinations of those SMOs and we can safely use it without fearing additional overhead when combining SMOs. The other combinations behave similarly; the respective charts are included in Appendix C.

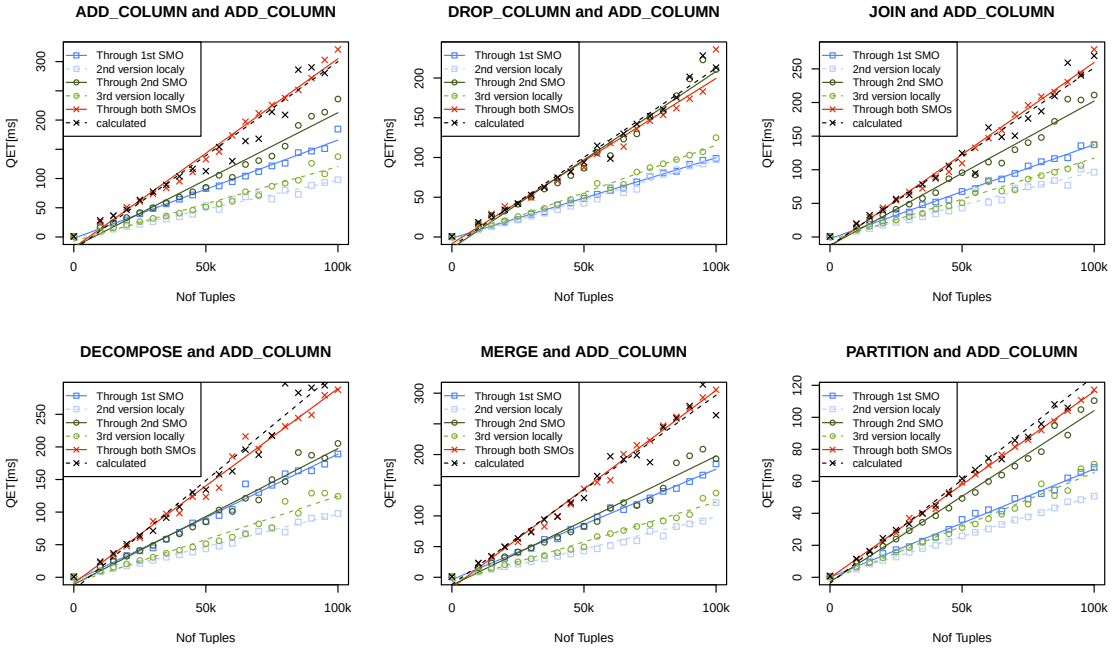


Figure 7.5: Scaling behavior for the combination of any SMO with the `ADD COLUMN` SMO.

Writing: Conducting the same analysis for write operations (insert, update, and delete) reveals that the overhead of write operations also **combines linearly** which facilitates stable estimations of the actual costs. Figure 7.6 shows the execution time of insert operations in three exemplary schema versions that are connected by two `ADD COLUMN` SMOs. The execution time is close to constant w.r.t. the number of tuples, so it is mainly determined by the execution of the stored procedures for propagating the data. We see the same pattern as before: data accesses become more expensive if and only if we propagate data backwards—from the evolved version with the added column back to the version without the added column—and the data is exclusively materialized in the old table version. The reason is that this constellation requires to store the values of the added column in an additional auxiliary table, which increases the required time. For writing at the third schema version with only the first version being materialized, we face this overhead two times; once for each add column SMO, which explains the doubled overhead. The slight difference in the actual overhead is caused by the different table sizes, since the third table version has one column more than the second table version and two columns more than the first one.

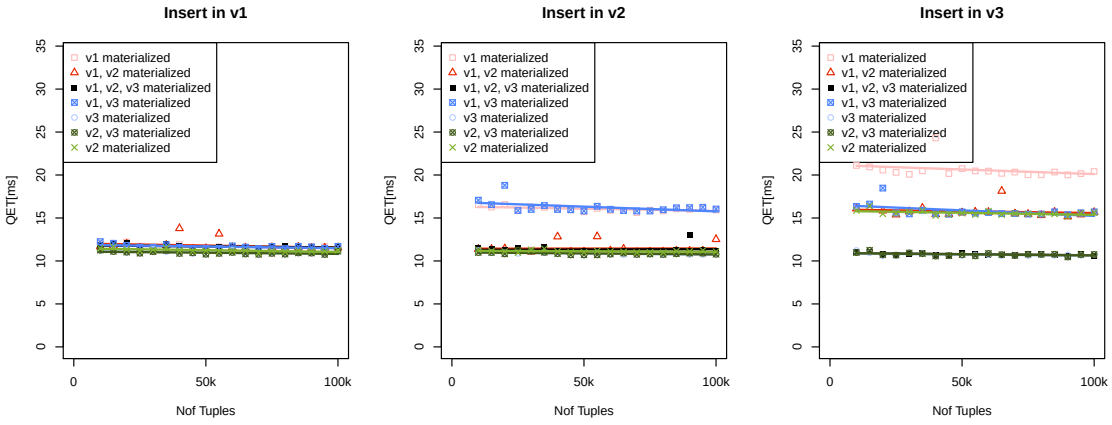


Figure 7.6: Insert operations in an evolution with two subsequent ADD COLUMN SMOs.

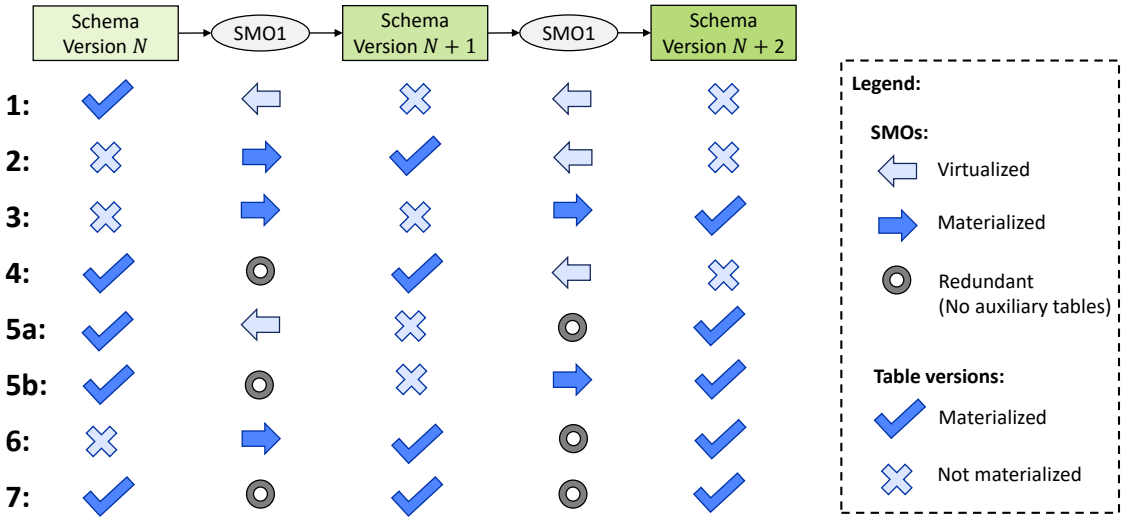


Figure 7.7: All materialization states for evolutions with two SMOs.

We conducted the same measurements for all combinations of SMOs and empirically confirmed that the execution time of write operations through sequences of SMOs is also **easily computable** by **linear combination** of the respective execution times of the propagation through single SMOs. In the evaluation, we show that this assumption is stable enough to obtain very precise estimations. The execution time of single write operations is influenced less by the number of tuples in the accessed tables but is dominated more by an almost constant overhead for propagating the single write operation through an SMOs. The same holds for longer sequences of SMO. The costs for writing add up linearly, which makes perfect sense, since write operations are propagated stepwise with stored procedures along SMO and simply call the stored procedure of the respectively next SMOs (cf. Section 6.3).

In sum, we have shown that the costs for the propagation through two SMOs can be precisely calculated as a linear combination of the costs for the propagation through single SMOs. Reading along SMOs usually requires joins with auxiliary tables and writing requires a stored procedure call. Both things are independently of other SMOs and explain the linear combination of the costs.

7.3.3 Data Access Patterns

For a deeper understanding and evaluation of the possible materialization states, Figure 7.7 shows all possible materialization states for sequences of two SMOs. We consider all possible materializations of the table versions: each table versions can be materialized or not—however, at least one table version must be materialized in order to not lose all the data. For most configurations of materialized table versions it is straight forward to determine in which direction data accesses are propagated through the SMOs. There is one exception: On the Lines 5a and 5b, both the first and the last schema version are materialized. This means that read operations on the middle schema version are either propagated to the first one (Line 5a) or to the last one (Line 5b). Since read operations are propagated to exclusively one side, there is no need to maintain auxiliary tables for both SMOs. In a specific scenario the decision depends on the cost model itself: we choose to propagate read accesses through the SMO that entails the least overhead for the current workload. Given a set of materialized table versions for a multi-schema-version database system, this pattern allows to derive the materialization states of all the SMOs in the schema versions catalog. The materialization states of the SMOs determine whether there are auxiliary table for reading forwards (materialized) or backwards (virtualized), or whether there are no auxiliary tables at all when data is never propagated through the SMO (redundant SMO).

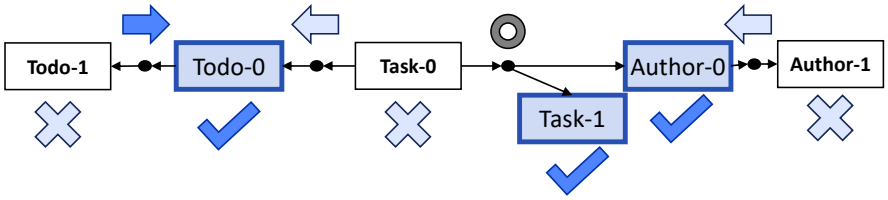


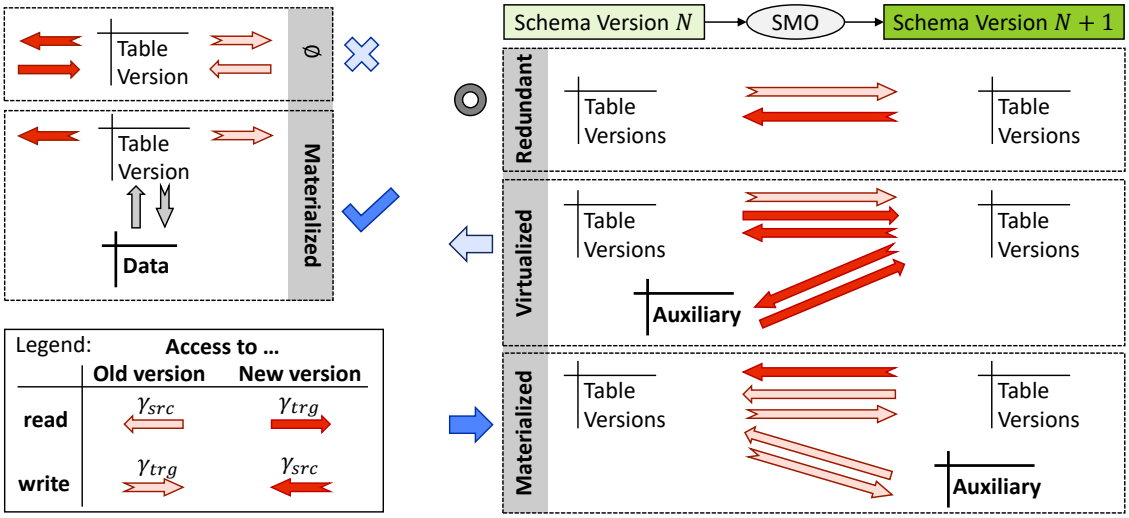
Figure 7.8: Materialization states for the TasKy example in Figure 7.1(c) on page 102.

In the TasKy example with the materialization shown in Figure 7.1(c) on page 102, we materialize the table versions `Todo-0`, `Task-1`, and `Author-0`. In Figure 7.8, we take the same materialization states of the table versions and summarize the resulting materialization states of all SMOs. We make sure that every single table version can propagate its read and write accesses to a materialized table version. This also holds for those table versions that are not explicitly accessed in the workload, which contains all table versions that are intermediate steps between schema versions but not part of any schema version. Even though those table versions are not explicitly accessed by users, the propagation of write operations requires to read these intermediate table versions as well.

Obviously, all materialized table versions can directly access the physical data but accesses to all the non-materialized table versions need to be propagated to the materialized table versions. In that sense, read operations on the table version `Todo-1` are propagated to access table version `Todo-0`. As a consequence, the `DROP COLUMN` SMO is virtualized, which means that data is stored only at the source side in `Todo-0` plus potential auxiliary tables needed for data access at the target site. As can be seen from the semantics definition of the `DROP COLUMN` SMO in Appendix A.2, there are no auxiliary tables required for source-side materialization, so we are lucky in this case. Equally, the `RENAME COLUMN` SMO for accessing `Author-1` is also virtualized to map data accesses from `Author-1` to `Author-0`. To access table version `Task-0`, either the `PARTITION` or the `DECOMPOSE` SMO need to be materialized which includes storing additional auxiliary tables to read and write the `Task-0` data correctly without any information loss. Since the analysis of single SMOs shows that the overhead of the `PARTITION` SMO is smaller, we materialize the `PARTITION` SMO and never use the `DECOMPOSE` SMO for reading.

Figure 7.9 summarizes the generalized materialization states for table versions and SMO; and it depicts the respective access pattern that determine the actual costs for the data access. In Figure 7.9(a), we see that there are two materialization states for table versions: materialized and non-materialized. When the table version is materialized, its data is stored physically, so read operations can be executed locally. Write operations are not only executed locally, but always propagated through all incoming and outgoing SMOs since other materialized table versions or auxiliary tables can be affected by the write operation as well. For the same reason, we also propagate write operations to non-materialized table versions. Read operations on a non-materialized table version are only propagated through either the incoming or one outgoing SMO to the cost-wise nearest materialized table versions.

Given the set of materialized table versions, we already discussed how to determine the materialization state of SMOs and we identified three possible states: materialized, virtualized, and redundant. Figure 7.9(b) summarizes these three states and depicts the respective access propagation pattern. Since redundant SMOs are never used for propagating a read operation, there are no auxiliary tables required. No information will be lost since both the source and the target side can be read without propagating data through the SMO. Merely write operations need to be propagated through the SMO to ensure global consistency among all table versions. Virtualized SMOs answer read operations on the target side by reading from the source side and consolidating the data with the respective auxiliary tables. The same applies to write operations at the target side. Read operations on the source side are never propagated through the virtualized SMO. Merely write operations are propagated to the target side as well, however, they will not affect the auxiliary tables of the SMO. Nevertheless, it is important to propagate the write operations further through the schema versions catalog, since there might be further materialized table versions or auxiliary tables to be updated as well. For a materialized SMO, the same pattern applies vice versa. In sum, we are now able to determine the materialization states of SMOs and the access propagation patterns based on a given set of materialized table versions.



(a) Table version states.

(b) Full replication to all schema versions.

Figure 7.9: Access patterns for different materialization states.

7.3.4 Learned Cost Model

From the previous performance study, we got the following four insights that form the basis for the adviser's cost model. First, we saw that the query execution times for both reading and writing **scale linearly**—some write operations even constantly—with the number of tuples. Second, both read and write operations are **more expensive, when auxiliary tables are involved**. Third, the query execution time of any operation **never decreases** with the number of SMOs through which it is propagated, so the cost model is **monotone** w.r.t. the access propagation through chains of SMOs. Last but not least, the costs for propagating data through multiple subsequent SMOs are the **linear combination** of the overheads for propagating data through the single SMOs. Hence, there is no additional cost for combining SMOs, neither is there a significant benefit.

We now condense this knowledge into a cost model, which is the basis for InVerDa's adviser. The cost model returns the estimated costs for executing a given operation on a given table version with a given materialization schema. These costs are mainly determined by the resulting data access pattern as discussed in the previous section. Summarizing the possible access propagation patterns from Figure 7.9(b), we see that there are four corresponding access patterns as shown in Figure 7.10: (1) forwards without auxiliary tables, (2) backwards without auxiliary tables, (3) forwards with auxiliary tables, and (4) backwards with auxiliary tables. For each of these four cases, the cost model provides a function of the form $a_{op}^{SMO}x+b_{op}^{SMO}$ for each combination of an SMO and an operation, with x being the size of the table in bytes. Besides the costs for data propagation through SMOs, we also need a cost model for reading and writing materialized table versions locally, which is also of the form $a_{op}x+b_{op}$ and globally applies to any table version. Since the specific parameters a and b may greatly differ depending on the used hardware, DBMS, etc., we do **learn the parameters individually** for each system. The scaling behavior of the propagation overhead follows a linear function. In order to learn the cost model, we create exemplary scenarios with one SMO respectively and a growing number of tuples, measure the propagation overhead, and use linear regression to learn the parameters a and b of the linear cost models.

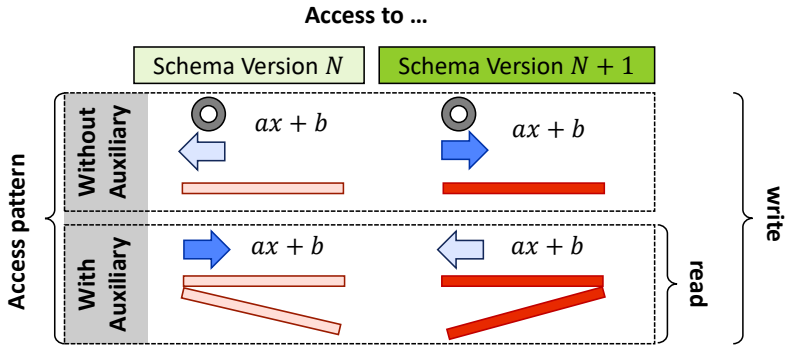


Figure 7.10: Cost model for the different access patterns.

The learned functions for the performance of access propagation through single SMOs are the basis for implementing the cost function $\mathcal{C}(c, \mathcal{M}, op)$, which returns the estimated costs for executing the operation op on the table version c with a given set of materialized table versions $\mathcal{M}$. With the given cost models for the access propagation through single SMO, we can estimate the costs for any given workload and materialization schema since the overheads combine linearly. The estimated costs are determined as follows:

1. Given a set of materialized table versions $\mathcal{M}$, we determine the materialization states of the SMOs. This step contains a nested optimization problem: for each table version, we determine the cost-wise closest materialized table versions to execute read operation as cheap as possible with the given materialization. Since the cost function is monotone w.r.t. the length of the propagation path, we can use the Dijkstra Algorithm to find the cheapest data access path for each single table version. The Dijkstra Algorithm uses merely the cost model for the propagation through chains of SMOs, so there is no cyclic dependency between the shortest path search and the overall cost estimation for a given materialization.

2. We calculate the costs for each operation op on each table version c in the workload separately using the discussed cost function $\mathcal{C}(c, \mathcal{M}, op)$. The costs are computed as the sum of the estimated costs for all involved SMOs and table versions. Read operations are directed to materialized table versions on the cheapest path. Write operations are propagated through the whole schema versions catalog since auxiliary tables might be affected anywhere.

3. We sum up these single costs and weight them according to the given workload $\mathcal{WL}(c, op)$.

This cost model facilitates to estimate the costs of a given workload for a given materialization, which allows to implement a cost model-based optimizer that proposes the best materialization to the DBA.

To sum up, we analyzed the quantitative and qualitative characteristics of the data access propagation in detail. Among other insights, we found that the overheads for data access propagation through single SMOs combine linearly when we chain up several SMOs to an evolution. This allows to precisely estimate the costs for accessing data with a given materialization with an error of less than $10\,\%$ for short evolutions, as we will show in the evaluation. Most importantly, the learned cost model allows to estimate the costs orders of magnitude faster than measuring the actual costs after migrating the database physically, as we will show in the evaluation as well. This is an essential prerequisite for cost model-based optimization algorithms. In the next section, we define the search space of all potential materializations—another key ingredient for InVerDa's adviser.

7.4 SEARCH SPACE

To find the best physical materialization, we have to know the whole search space of possible materializations in the first place. Intuitively speaking, every single table version in the evolution history can be either materialized or not materialized, which yields $2^{|\mathcal{TV}|}$ possible materializations, with $\mathcal{TV}$ being the set of all table version—however, not all these materializations are valid. When e.g. not a single table version is selected to be materialized, then the data will be lost. In this section, we first define the set of valid materialization schemas $\mathcal{MATS}$ and then estimate the bounds of the search space's size.

We consider a **valid materialization** to ensure that each single table version $c \in \mathcal{TV}$ can propagate its data access to materialized table versions. In this case, we say that the respective table version c is **covered**. A table version $c \in \mathcal{TV}$ is covered by a materialization $\mathcal{M}$—this means that $\text{isCovered}(c, \mathcal{M}) = \top$ holds—when there exists a path through the schema versions catalog where the materialization states along the path aggregate to $\top$: The aggregation of the materialization states is disjunctive along a sequential path and conjunctive if the path splits at an SMO with two source or two target table versions. Such paths may even take steps contradicting the direction of the SMOs.

$$\text{isCovered}(c, \mathcal{M}) = c \in \mathcal{M} \vee \left(\bigvee_{e \in \mathcal{E}(c)} \left(\bigwedge_{c' \in \mathcal{N}(e,c)} \text{isCovered}(c') \right) \right) \tag{7.4}$$

By enforcing this condition for all table versions of the schema versions catalog, we ensure that the whole data of all table versions is represented at least once in the materialized schema. Hence, a materialization $\mathcal{M}$ is valid if the following condition holds:

$$\forall_{c \in \mathcal{TV}} \text{ isCovered}(c, \mathcal{M}) \tag{7.5}$$

Building upon this validity check, we define the set of all possible materialization states $\mathcal{MATS}$ as:

$$\mathcal{MATS} = \{\mathcal{M} | \mathcal{M} \subseteq \mathcal{TV} \wedge \forall_{c \in \mathcal{TV}} \text{ isCovered}(c, \mathcal{M})\} \tag{7.6}$$

The resulting **search space** of possible materializations can **grow exponentially** with the number of table versions in the evolution history. The actual scaling behavior of the search space depends on the graph structure of the schema versions catalog, as we will discuss in the following. A multi-schema-version database system with N created table versions, which are never evolved, can be persisted in exactly one way: all created tables must be materialized. If each of these N table versions evolves with one SMO that e.g. adds a column, we have N independent evolution with the length of one SMO each, which creates $|\mathcal{TV}| = 2N$ table versions. For each of the N independent evolutions, the data can be persisted in the source, or in the target, or in both table versions, which allows 3^N materializations in total. Further, a longer evolution with M SMOs that all depend on each other by e.g. adding M columns to one table also allows $2^M - 1$ possible materializations, as any non-empty subset of all M table versions can be materialized. Hence, both multiple independent evolutions as well as sequences of dependent evolutions cause an exponential search space size and this complexity remains when evolutions of single tables are merged or split by further SMOs.

In sum, we defined the set of all valid materializations $\mathcal{MATS}$ and estimated its size $|\mathcal{MATS}|$. This search space size depends on the structure of the schema versions catalog graph and is not necessarily exponentially large, however, even the discussed simple and common scenarios, such as adding multiple columns to one table, have an exponentially growing number of potential materializations. Hence, we can safely assume that the search space will grow exponentially with the number of table versions in many practical scenarios, which calls for sophisticated optimization algorithm that do not enumerate the whole search space. To provide a generally applicable adviser we tailor the optimization algorithm accordingly to a potentially exponentially large search space in the next section.

7.5 OPTIMIZATION ALGORITHM

As the search space of physical materializations potentially has an exponential size w.r.t. the number of table versions in the evolution, there is no feasible way to fully explore all possible materializations and compare their estimated costs to find the best solution. While there are merely 59 possibilities for the six table versions in TasKy, there are up to 1×10^{61} possibilities for the 203 table versions in the Wikimedia Benchmark [40] making it impracticable to enumerate the whole search space. Instead, we can only approximate a good solution. Unfortunately, there is no reliable way to use the presented cost function as a heuristic to decide whether a specific table version should be materialized or not, because the cost function can only be evaluated globally and cannot assign costs for materializing one specific table version locally. Further, the cost model does not behave monotone when extending or reducing a materialization $\mathcal{M}$, since the actual access propagation paths may change significantly when table versions are added to or removed from the materialization. Therefore, we cannot assume that an optimal solution is built upon optimal solutions for subproblems.

The cost function is only **globally evaluable** for a whole given materialization. It is not possible to determine the local costs for materializing or not materializing a single table version, since these costs always depend on the materialization states of all neighboring table versions. Since read operations are propagated to the cost-wise closest materialized table version, the cost function contains a nested optimization problem of finding the best propagation path for each table version. Therefore, the objective of the optimizer is to solve a bi-level optimization problem with a local optimization of the shortest access paths and a global minimization of the aggregated local costs [49]. The adviser finds the optimal access path for each table version using the Dijkstra Algorithm. On the one hand side, the costs for not materializing a specific table version c can be very high assuming this requires to propagate data through an expensive neighboring SMO. On the other hand side, these costs may fall almost down to zero if we merely materialize another table version c' that was e.g. created by renaming a column in table version c. In sum, the cost function estimates the costs of a materialization as a whole with no means to assign costs to single table versions or SMOs.

Further, when enumerating the search space by e.g. stepwise extending the materialization by newly materialized table versions, the results of the cost function are **not monotone**. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two different materializations from the set of all valid materializations $\mathcal{MATS}$. We can neither assume that $\mathcal{C}(c, \mathcal{M}_1, op) \leq \mathcal{C}(c, \mathcal{M}_1 \cup \mathcal{M}_2, op)$ holds nor can we assume that $\mathcal{C}(c, \mathcal{M}_1, op) \geq \mathcal{C}(c, \mathcal{M}_1 \cup \mathcal{M}_2, op)$ holds. Since the workload mixes read and write operations any additionally materialized table version can either increase the overall costs as writes become more expensive or costs can be reduced as reading requires less propagation.

As a consequence of the global cost function and non-monotone behavior of extending materializations, the optimization problem cannot be divided in smaller subproblems. Hence e.g. dynamic programming algorithms are not an option and the adviser is forced to explorer every corner of the search space. This is not possible in practice due to its potentially exponential size. Also greedy optimization algorithms will return solutions that are far from optimal—in the evaluation, we will see that the best materialization is significantly better than intuitive solutions that could be returned by greedy algorithms. A common approach for such optimization problems is to use **evolutionary optimization algorithms** that maintain a population of the currently best solutions and randomly evolve them by small manipulations but also by more significant mutations [22]. According to Darwin's Law, only the best solutions survive and are evolved further to find better and better solutions. While there are no guarantees that the adviser will find the globally optimal materialization, evolutionary algorithms have shown very good results currently and are the best option to solve optimization problems as the one that InVerDa's adviser faces [49].

The evolutionary optimization algorithm of InVerDa's adviser initially creates a set of reasonable materializations as the **initial population**. This covers non-redundant materializations of the oldest and newest table versions as well as full materializations of all table versions. Further initial solutions can be obtained **greedily** by materializing the most accessed table versions first and then incrementally adding more table versions to the materialization until the space budget is used. As **evolution steps** we use fine-grained fuzzy modifications that move or copy or merge the materialization of a table version through a neighboring SMO to/with the next table versions. As coarse-grained mutations, we randomly select table versions and change their materialization states. Starting with the initial materializations, we use the discussed evolution steps to generate new materializations in the population. The presented cost function can efficiently estimate the costs for a given materialization; following the idea of evolutionary optimization algorithms, we remove those materializations with the highest estimated costs from our population and continue the optimization until we reach a given maximum number of considered materializations or do not see any improvement for a certain time.

In sum, InVerDa's adviser optimizes the materialization using an evolutionary algorithm. We decided for the evolutionary optimization, because the cost function behaves non-monotone when adding or removing table versions from a materialization, which makes other algorithm like dynamic programming inapplicable. The proposed optimization algorithm starts with a set of intuitively promising materializations and further evolves them with small modifications and bigger mutations to find increasingly better materializations. In the evaluation, we will show that this evolutionary algorithm finds the optimal solution for small examples such as our TasKy example from Section 2.3 and that it achieves significant improvements for long evolution histories such as the evolution of Wikimedia.

7.6 EVALUATION

We have presented the problem of finding a good materialization in the potentially exponential search space of materializations in a multi-schema-version database system and proposed a cost model-based adviser that returns an optimized materialization for the given workload on a specific system. Now, we empirically analyze this adviser focusing both on the cost model (Section 7.6.1) and on the actual optimization algorithm (Section 7.6.2). We conducted the evaluation on a PostgreSQL 9.4 database on a Core i7 machine with 2.4GHz and 8GB memory.

7.6.1 Cost Model

We analyze scenarios that cover all evolutions with two subsequent SMOs as well as the TasKy example to obtain a detailed picture of the strength and weaknesses of the learned cost model.

Evolutions with two SMOs: For this evaluation, we consider all evolutions with two subsequent SMOs. Thereby, the first SMO always creates a table version $R(a, b, c)$. The second SMO further evolves this table $R(a, b, c)$, so we end up with three schema versions just as used before (Section 7.3.2). We execute read and write operations on the third schema version with the first being materialized, to obtain the costs for propagating the data access through the two subsequent SMOs. We also estimate the costs for this propagation using our learned cost model (Section 7.3.4) to compare them to the actually measured costs.

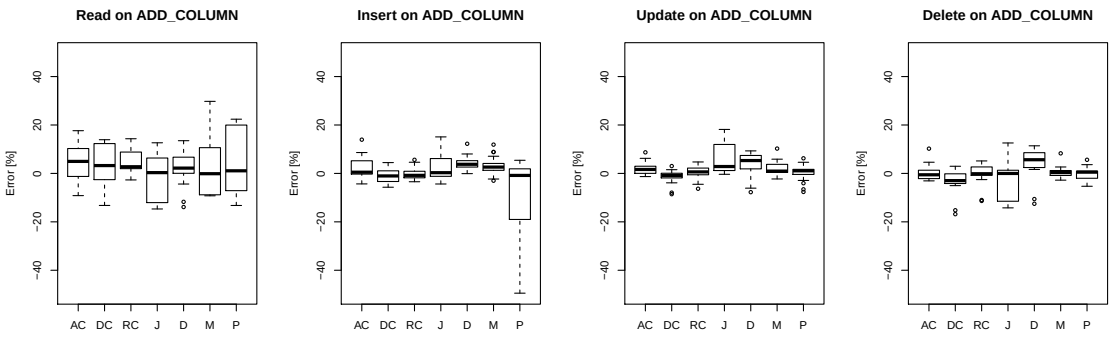


Figure 7.11: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being an ADD COLUMN SMO.

Figure 7.11 shows the estimation error, which expresses the percentage of the misestimation compared to the actually measured costs. In the figure, we show the estimation error for all scenarios that have an ADD COLUMN as second SMO while the respective first SMO is shown at the x-axis of the box-plot charts. For this first SMO of the evolution, we use the following abbreviations: AC (ADD COLUMN), DC (DROP COLUMN), RC (RENAME COLUMN), J (JOIN), D (DECOMPOSE), M (MERGE), and P (PARTITION). The deviation shown in Figure 7.11 covers 50 different table sizes from 5000 to 100 000 tuples. As can be seen, the mean error is very low in the range of 5 %, which is a great basis for the optimizer. Also the deviation is within reasonable bound. The write operation are precisely estimated in almost all scenarios. The deviation for read operations is roughly 10 %, which is a bit higher but still a reasonable basis for the optimizer. As comparison: Typical errors for the cost estimation of a query optimizer in a common DBMS are two to ten times higher and still allow the optimizer to find good plans [118].

Figure 7.12 shows the results of evaluating the same setup for all evolutions with a JOIN as second SMO, which basically confirm the behavior we saw for the ADD COLUMN SMO. The same applies to the evolution with MERGE as second SMO as shown in Figure 7.13. Merely, the deviation for evolutions with the MERGE SMO, is higher, which we account to the number of auxiliary tables (two to five) that need to be managed for this SMO. Nevertheless, the mean estimation of all scenarios are very accurate and allow the optimizer to safely rely on the cost model, which can be evaluated way faster than actually migrating and analyzing the database physically. We exemplarily choose the ADD COLUMN, the JOIN, and the MERGE SMO to validate the feasibility of the cost model. The evolutions with the remaining SMOs behave similarly, which is not surprising as they are the inverses of the depicted SMOs. The respective measurements are summarized in Appendix D.

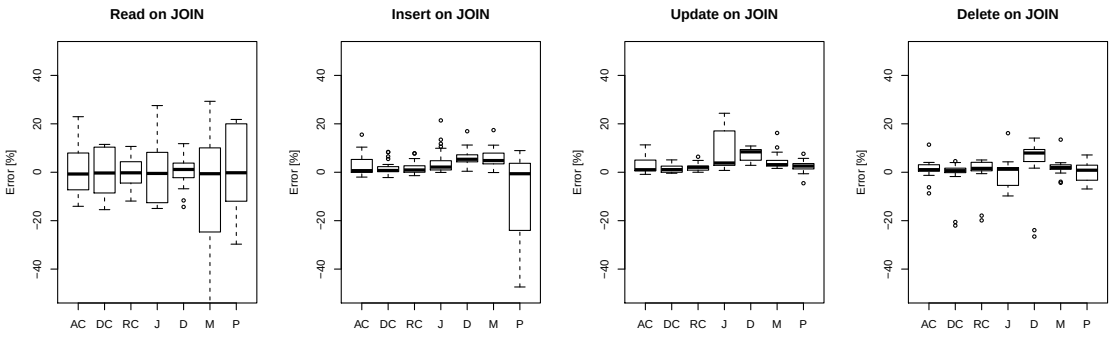


Figure 7.12: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a JOIN SMO.

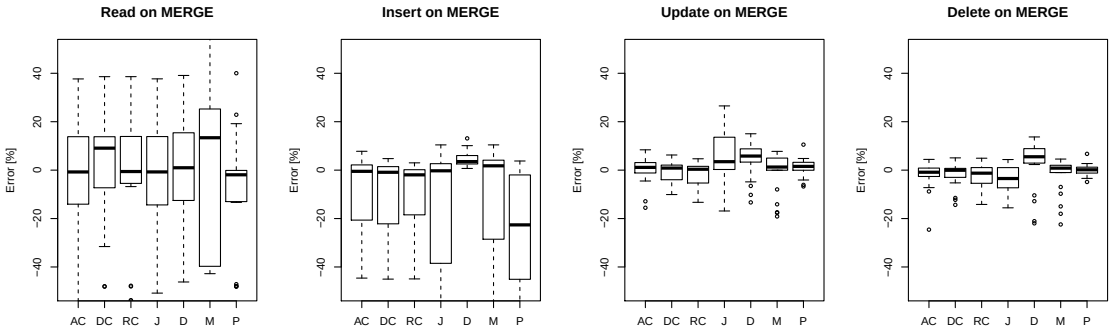
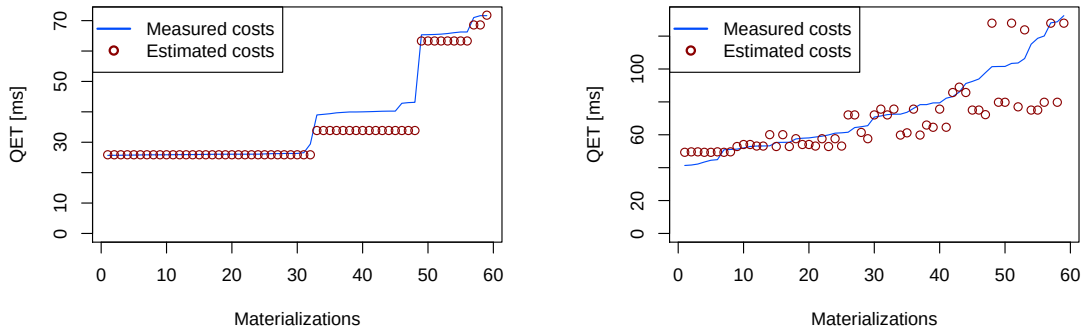


Figure 7.13: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a `MERGE` SMO.



(a) Error of reading in Do!.

(b) Error of workload mix.

Figure 7.14: Analysis of the estimation errors in the TasKy example.

TasKy: Now, we apply the cost model to the TasKy example (Figure 2.4, page 28) as a small realistic scenario. Figure 7.14(a) shows the estimated and the measured costs for reading the Do! schema versions for all possible materializations. In fact, the TasKy scenario allows 59 different materializations ranging from non-redundant materializations over partially redundant to the fully redundant materialization. In the shown figure, we sort the materializations by their actually measured costs. As can be seen, the cost model estimated the actual costs for reading the Do! schema version almost perfectly; the mean estimation error is 14 %. Most importantly, the order of the estimated costs for the different materializations is equal to the order of their actual costs, which allows the adviser to make very precise decisions only based on the cost model and without the need to physically migrate the database to evaluate specific materializations individually. Figure 7.14(b) confirms these finding also for the mixed workload that we presented in Figure 7.1 on page 105. The deviation is 24.2 % but as can be seen, the general order of the materializations is still estimated well. Thus, whenever the adviser determines a good materialization based on the cost model, chances are very high that this materialization will also perform well in practice.

Since the optimization algorithm needs to cover a very large search space, the time for calculating the costs of each specific materialization must be as short as possible. We measure the time required for the cost model-based estimation in comparison to physically migrating the database to a new materialization and measuring the actual costs for the given workload and the given materialization. Exemplary, we assume the database to materialize exclusively the initial TasKy version with 100 000

entries; to get the costs for executing a workload mix with Do! being materialized, we need $8.21\,\mathrm{s}$ to change the materialization and another $33.32\,\mathrm{s}$ to measure the mean execution time for a read/write workload mix on all schema versions. This sums up to $41.53\,\mathrm{s}$ for the whole example. In contrast, the cost model returns the estimated costs after $132.56\,\mu\mathrm{s}$, which is a speedup of factor 2.5×10^5. With growing table sizes, the time for physically migrating and analyzing a database will grow even further and can easily last hours or even days. At the same time, estimating the costs using our learned cost model is independent from the size of the tables and thereby has constant time requirements. The cost model merely requires an initial effort to learn the parameters for the actual system. For this purpose, we create exemplary tables with a growing number of tuples—the cost model used in the evaluation is trained with 1000 to 100 000 tuples in 50 steps, which took less than ten minutes. A higher precision of the cost model can be achieved by increasing the size of the training tables, hence the DBA has to determine the preferred tradeoff between initialization time and accuracy. In conclusion, the cost model proved to be reasonably precise to allow the adviser profound decisions. Using the cost model is orders of magnitude faster than actually measuring the costs for all the different materializations separately.

7.6.2 Optimizer

The objective of the optimizer is to find the best possible materialization for a given workload on a given system. Since the search space can grow exponentially large with the number of table versions in the multi-schema-version database system and the cost function can only be evaluated globally, we decided for an evolutionary optimization algorithm in Section 7.5. Now, we will run this optimization algorithm both in the TasKy scenario and in the Wikimedia scenario to evaluate its feasibility.

TasKy: The TasKy example is comparably small, which allows us to enumerate all possible materializations for the evaluation. This provides insights into the whole search space and allows to analyze the most beneficial materializations in detail. Figure 7.15 shows the results for running InVerDa's adviser for 21 different workloads. The workloads are structured into two dimensions. First, users can access any non-empty subset of the three schema versions Do!, TasKy, and TasKy2 as depicted in the vertical dimension of Figure 7.15. Second, the workload can comprise (1) only read operations, (2) a $50:50$ mix of read and write operations, or (3) only write operations as represented in the horizontal dimension of the figure. Obviously, the 21 possible workloads that result from combining the two dimensions are merely representative corner points in the space of all possible workloads, however, they serve well to understand the potentials and benefits of InVerDa's adviser. Additionally, we will validate the findings with a more unstructured workload according to Figure 7.1 on page 105.

For each workload, we have depicted four measured values in Figure 7.15. The first bar represents the best possible materialization; since the search space is small, we can fully enumerate it and determine the globally best materialization, which should be found by the adviser. Since the adviser relies on the learned cost model with its small deviations, it might happen that the materialization proposed by the adviser (represented by the second bar) may have slightly higher costs than the optimum. The third bar shows the costs of the best non-redundant materialization to analyze the benefits of allowing partially or fully redundant materializations as well. Finally, the fourth bar shows the costs of the worst possible materialization for comparison. Below the bar chart with the costs of the four discussed materializations, we deep dive into the optimal materializations. As explained in the legend, we show an abstraction of the schema versions catalog with all six table versions. We manually analyze both the best non-redundant and the best redundant materialization and summarize which table versions should be materialized.

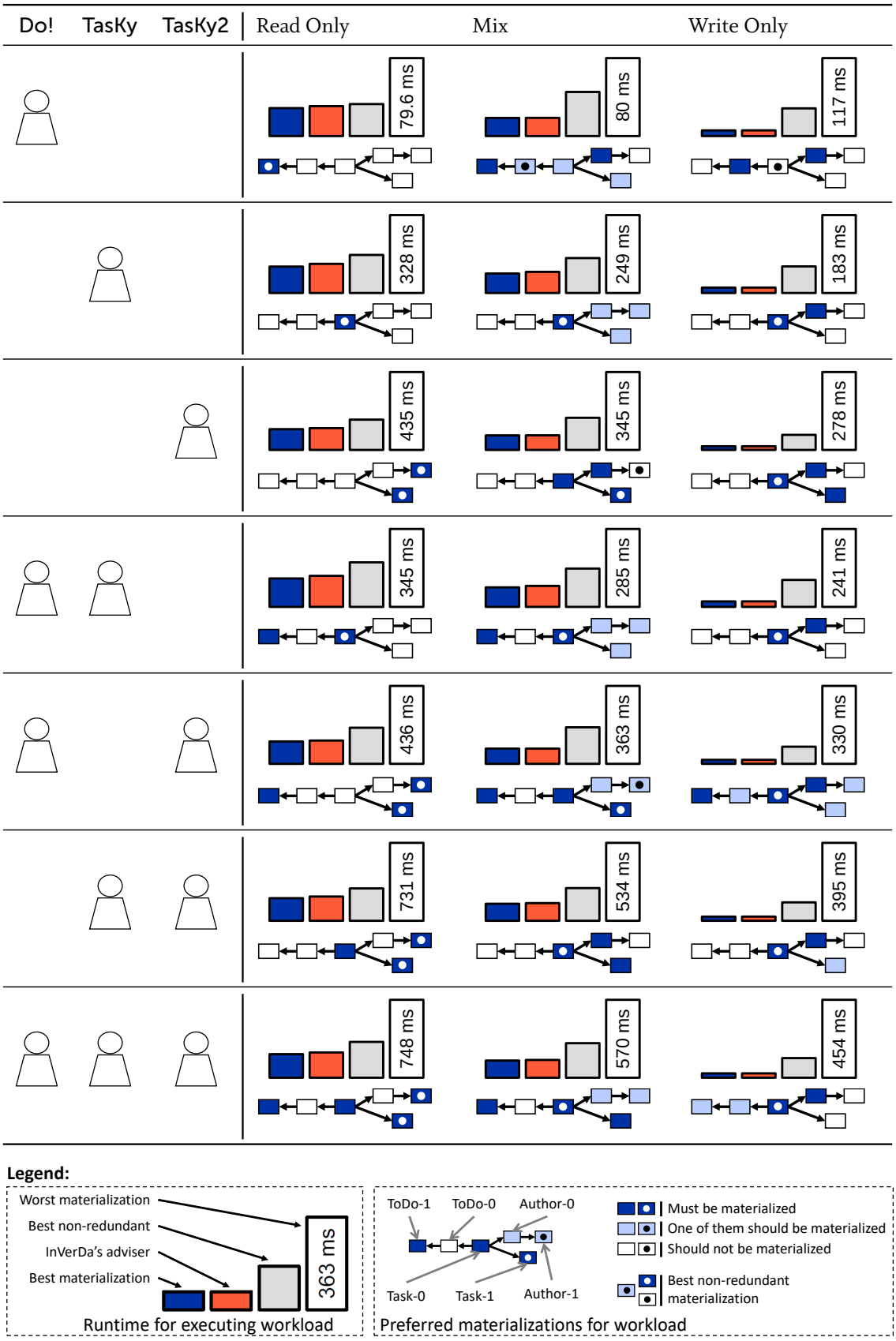


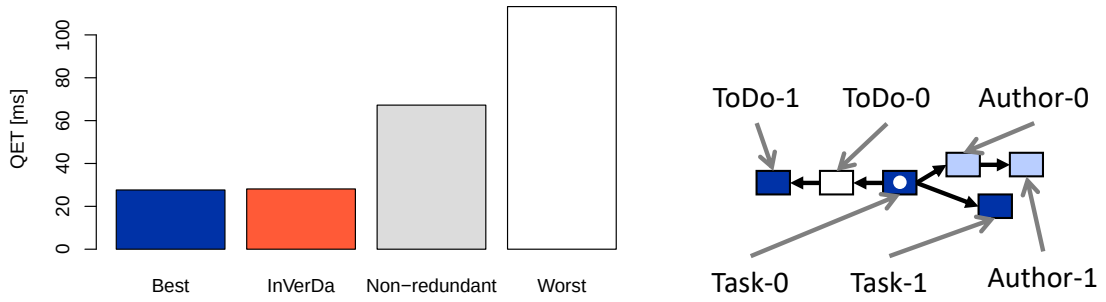
Figure 7.15: Adviser results for **TasKy**.

There are many interesting insights to discover in the analysis shown in Figure 7.15. First and foremost, there is a **significant optimization potential**, since the worst materialization is always significantly more expensive than the best materialization. For read operations the optimization potential is up to factor 5. Write operations cause a higher overhead which makes a feasible materialization even more important. As can be seen, the worst materialization can be up to 30 times more expensive than the best one. Fortunately, InVerDa's adviser always proposes a materialization that is comparably fast as the best materialization; in fact it is never more than 10 % slower than the optimum. We conclude that the **cost model works** for the given scenarios and that the performance for the proposed materialization is actually as good as estimated. Even though the deviation of the cost model causes the adviser sometimes to pick a materialization that is slightly behind the best possible one, the cost model-based optimization allows to investigate orders of magnitude more materializations in the same time, which clearly compensates the deviation in most practical scenarios.

Further, the best non-redundant materialization is up to factor 10 more expensive then the partially or fully redundant materializations found by InVerDa's adviser. Hence, we conclude that allowing redundancy enables further significant performance improvement; for longer evolutions this effect will even increase. Even though it might be counter-intuitive at first glance that especially write operations can benefit from redundant materializations, there are good reasons for this: partial redundancy can reduce the number of auxiliary tables that need to be updated during writing. This underlines that simply using the **naïve materialization** is often significantly **more expensive** than a materialization proposed by InVerDa's adviser.

Let us analyze the best materializations in more detail starting with workloads that are restricted to one single schema versions (first three rows of Figure 7.15). When reading only, we obviously materialize the accessed table versions, which allows to answer all read queries locally. There is no need to redundantly materialize further table versions since they would not be accessed anyhow. For workloads that include write operations, the best materializations are not that obvious to find, which supports the need for an automated adviser. For pure write workloads, the best non-redundant materialization always materialized the initial TasKy table version. The reason becomes clearer, when zooming into the involved SMOs. The propagation of write operations from the Do! schema is cheap since the virtualized `PARTITION` SMO requires less auxiliary tables than the materialized one—the virtualized `ADD COLUMN` SMO even works without any auxiliary table. Therefore, less write operations on auxiliary tables need to be computed and executed. The same applies to the propagation of write operations from the TasKy2 schema version. When redundancy is allowed, the best materializations use partial redundancy to further reduce the number of write operations on data and auxiliary tables.

For the workloads that mix data access at multiple schema versions (last four rows of Figure 7.15), we see similar patterns. Obviously, read accesses perform best when the accessed table versions are materialized; since no data needs to be propagated between schema versions, there is no need to materialize further table versions. When space is bound to a non-redundant materialization, read accesses perform best, when the data is materialized according to the TasKy2 schema version—at least when TasKy2 is accessed, otherwise the initial TasKy schema is materialized. This indicates that the propagation of read operations from TasKy to Do! and from TasKy2 to TasKy is more expensive than in the opposite directions. For writing, the best materializations are different: For the non-redundant materializations, it is always beneficial to materialize the initial `Task-0` table version, as discussed before, this is because the virtualized `DROP COLUMN`, `PARTITION`, and `DECOMPOSE` SMOs require the update of less auxiliary tables. The partially redundant materializations always contain the initial `Task-0` table version as well; however, they also materialize the `Author-0` table version, which simplifies updating auxiliary tables. Especially the partially redundant materializations for the mixed workloads include other tables as well, which are often not intuitively obvious since the underlying access propagation is not visible from a user perspective. The proposed partially redundant materializations are significantly better than the naïve non-redundant materializations. This again emphasizes the need for an adviser.



(a) Performance of different materializations.

(b) Best materializations.

Figure 7.16: Adviser results for workload mix from Figure 7.1, page 105.

So far, we considered rather homogeneous workloads along the two dimensions in Figure 7.15—the gathered insights also hold for more heterogeneous workloads. Let us again consider the workload shown in Figure 7.1 on page 105, where mainly the Do! schema version is used for reading tasks and removing them after completion, while users use the TasKy2 schema version to create and maintain their tasks. The initial TasKy schema version is assumed to be outdated and merely used by a handful of nostalgic users. Figure 7.16 summarizes the results of the same experiments we ran before but now with the heterogeneous workload. Figure 7.16(a) shows that our cost model works well, because InVerDa's adviser proposes a materialization that performs just as good as the optimum. As can be seen in Figure 7.16(b), the best non-redundant materialization materializes the initial `Task-0` table version—however, the partially redundant materialization proposed by InVerDa's adviser additionally materializes `Todo-1` as well as `Task-1` and one of the `Author` tables. The speedup gained by the partially redundant materialization is 2.5 compared to the non-redundant materialization. In sum, we confirmed that it is very beneficial to use InVerDa's adviser as it can achieve significant performance improvements that would be hard to achieve by a human DBA, since the performance of partially redundant materializations is usually hard to estimate manually.

Wikimedia: We now confirm our findings using the realistic scenario of 171 schema versions of Wikimedia. The 203 table versions allow a total of up to 1×10^{61} possible materializations, hence we can obviously not determine the overall best materialization. Instead, we will apply the presented evolutionary optimization algorithm in the Wikimedia scenario and analyze the achieved performance improvements in comparison to naïve and worst case solutions.

Figure 7.17 shows a run of the optimization algorithm: After 828 ms with 429 evolved members in the population, there was no significant improvement over the last 30 steps, so the adviser returned the so far best materialization. The figure depicts the estimated costs of the currently best materialization as well as the most recently added materialization in the population for each step of the evolutionary optimization. Further, it shows the actually measured costs for both the best initial materialization and the finally proposed one to validate the achieved performance improvement. The workload mixes read and write operations on schema version *v04619* (28th version) and *v25635* (171th version).

Due to the size of the search space, we do not know the best and the worst possible materialization, but we can safely say that InVerDa's adviser achieved an estimated speedup of factor 10 compared to the naïve initial materializations used by the evolutionary algorithm. Again, the estimations match the actual costs very well and we also observe a speedup of factor 10 when materializing the co-existing schema versions according to the adviser's proposal. Besides several small improvements, there are few steps that significantly improved the overall performance by introducing redundancy

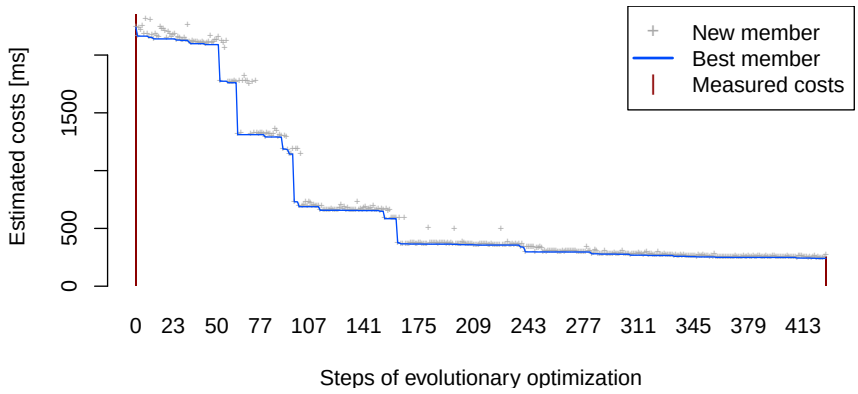


Figure 7.17: Adviser for Wikimedia.

to initially non-redundant members of the population. In other scenarios, it might be promising to search for better materializations longer—however, the Wikimedia evolution mainly uses `ADD COLUMN` SMOs, which are materialized already by the most promising initial population members. The actual optimization potential is merely to create partial redundancy along the evolution especially at those few SMOs that restructure existing table versions or drop columns. This already allowed the adviser to propose materializations with a significant performance improvement compared to the initial materializations—according to our detailed analysis, more heterogeneous evolutions provide an even higher optimization potential since the initial materializations are most likely worse.

In sum, we have shown that the cost model estimates the costs for running a given workload on a given materialization feasibly well. When the adviser decided for a materialization based on the cost model, it will also perform well in practice. Even though the cost model has slight deviations, its key advantage is the very fast calculation of the estimated costs within less than a millisecond. The fast evaluation of the estimated costs is crucial for the proposed evolutionary optimization algorithm. In fact, InVerDa's adviser has shown to approach the best materialization very close for short evolutions as our TasKy example. For long evolutions such as the Wikimedia scenario, we do not know the globally best materialization due to the size of the search space. However, we see that the adviser proposes materializations that perform orders of magnitude better than the initial naïve materializations. This makes the adviser an important and handy tool for DBAs, as they do not have to dive into the characteristics of data access propagation through SMOs but can simply optimize the materialization for the current workload with the click of a button.

7.7 SUMMARY

An adviser in a multi-schema-version database system, such as InVerDa, should propose the best materialization of the co-existing schema versions w.r.t. the current system, the current workload, and a given upper space limit. After formalizing all these ingredients, we defined the actual optimization problem, which is a bi-level optimization problem in a potentially exponentially large search space. To explore as much of this search space as possible within a reasonable time, we decided for a cost model-based optimization algorithm—as a foundations, we analyzed the performance characteristics of propagating data accesses in multi-schema-version database systems to define a stable cost model that can be individually learned for each specific system. Further, we defined the set of valid materializations, which completes the basics for the actual optimization algorithm.

We want the optimizer to globally minimize the costs of the overall workload on all table versions but for each table version we locally search its cost-wise shortest path to materialized table versions. Due to this characteristic, the cost function does not behave monotone when adding or removing table versions to/from the materialization. Therefore, we decided for an evolutionary optimization algorithm that handles such adverse conditions by design. In the evaluation, the proposed optimization algorithm has proven to reduce the costs by orders of magnitude compared to the naïve materializations, which would be hard to achieve for a human DBA. In conclusion, the adviser is of great support for DBAs, since they can achieve significant performance improvements even without understanding the complex characteristics of propagating data accesses through SMOs to materialized table versions and auxiliary tables.

FURTHER APPLICATIONS AND FUTURE WORK

InVerDa provides very powerful concepts and tooling to support database evolution within a relational database. So far, we focused on the conceptual contributions, particularly the creation of new schema versions by evolution or merging as well as an efficient materialization to let all the created schema versions co-exists at the same time within one database. However, InVerDa can be applied in many scenarios beyond this; some scenarios can be already fully covered with InVerDa, others require further research with InVerDa as a promising starting point. For each application scenario, we will discuss the problem with the current state-of-the-art, show how InVerDa can solve those problems, and discuss open research questions in order to achieve this goal. The application scenarios span a continuum from fully-supported by InVerDa to great research potential with InVerDa.

Particularly, we will use BiDEL's rich semantics to automate the database evolution in software product lines (Section 8.1) and utilize InVerDa's co-existing schema versions to support multitenancy application development (Section 8.2) as well as productions-test systems and rolling upgrades (Section 8.3). Further, we apply InVerDa's miX to simplify third-party-component co-evolution in big software systems (Section 8.4). All these scenarios are already well supported by InVerDa and require merely minor adjustments. Future extensions of InVerDa might cover zero downtime migration scenarios (Section 8.5). Finally, we detail on promising future research, which covers an extensions of BiDEL's expressiveness (Section 8.6) and performance tuning (Section 8.7).

8.1 SOFTWARE PRODUCT LINE ENGINEERING

Scenario with State of the Art: Almost every vendor of software systems faces the problem of many potential customers that all need more or less the same software but individually adjusted to specific processes, environments, and requirements. Creating an individual software product for every single customer by copying and adjusting the actual software, is a maintenance nightmare. Software product line engineering is a solution to this dilemma: different features are implemented separately within a software product line and a specific software product can be generated for a customer by merely selecting all desired features. As already discussed in the related work (Section 3.2), software product line engineering poses new challenges when it comes to the evolution of the database as there are at least two different dimensions of evolution: First, when customers create or change the configuration of their specific product, features are added or removed to/from the software product. The underlying database schema, which represents the consolidation of all the selected feature's database schemas, evolves as well since other data needs to be persisted, now. Second, when developers further evolve single features, their share of the database schema and all depending features may evolve, too.

Currently, it requires extensive manual work to create and evolve the database of each specific product from a software product line. Particularly, keeping existing data consistent and evolving it with the database schema is tough in practice since this breaks the beauty and simplicity of software product line engineering—there is more effort required than simply selecting the desired set of features. In research, there is upcoming tool support for these tasks [58], helping the developers in finding and resolving conflicts and inconsistencies during such evolutions. However, there is currently no solution facilitating non-expert users to simply evolve an individual software product, including the evolution of currently existing data, by only selecting the desired features in a graphical user interface.

Scenario with InVerDa: InVerDa provides all the functionality required for database evolution in software product lines. Assuming every implemented feature of the software product line includes a BiDEL-specified evolution script with only additive SMOs to create all tables and columns required to persist the data of the respective feature. To initially create the database schema for a selected set of features, all their individual evolution scripts are merged. Since they are purely additive and we can prevent naming conflicts by prefixing every name with the feature's identifier, the individual evolution scripts can be merged fully automatically.

InVerDa can also handle the database evolution occurring from changing a software product's configuration. The sequence of SMOs that describes the database evolution to the new version, can be easily derived from the SMOs of each involved feature: SMOs of newly selected features are added while the inverse SMOs are added for deselected features. Thus, BiDEL facilitates fully automatic database evolution support for software product lines without any interaction of the database developers.

Finally, InVerDa also covers evolving features and the resulting evolution of all products that contain the respective feature. BiDEL's SMOs explicitly capture the developers' intentions for a features evolution, which facilitates applying it to any running product. In case depending features build upon the old schema version of the evolved feature, we can use the merging functionality of InVerDa to co-evolve the database schema of the depending feature as well. In sum, the SMO-based database evolution supported with InVerDa can be used both for the initialization, for the evolution, and for the removal of product features in a software product line. This greatly simplifies database evolution in all the different dimensions of software product line evolution.

Future Work: Conceptually, InVerDa fully supports database evolution for software product lines. However, it is left for future work to validate this claim with an extensive case study. Further work might be required to create tool support for database evolution that is tightly coupled to the tooling of software product line engineering.

8.2 MULTITENANCY

Scenario with State of the Art: For many small companies, an own IT infrastructure would not pay off as it is underutilized most of the time; for such customers a software-as-a-service solution with an IT infrastructure that is shared among multiple clients is very appealing. Again, we have multiple similar applications that are individually adjusted for different customers. It has been empirically shown that it is performance-wise very beneficial to store the data of multiple customers within one database schema with shared tables, instead of creating an own database instance for every single customer [12]. Within those shared tables, there can be shared data visible to each and every customer—some master data like e.g. countries. Further, the shared tables have private individually added tuples that additionally carry the customer's identifier to not show this private data to any other customer. Since the individual data of all customers can be structured in different schemas, there are shared tables for the common core that all customers share as well as additional universal table, or pivot tables, or common extension tables, or tables for chunks of data that allow to adjust the database schema individually for each customer [11].

A major drawback of these approaches is that the individual tables and columns typically follow a very generic layout without typed columns that are often even unnamed, which implies a significant performance overhead as established optimizations cannot be applied. Further, the adaptations of the individual schemas are limited to adding tables and columns—more fundamental schema changes like merging, partitioning, or joining are not supported at all.

Scenario with InVerDa: With InVerDa, the developers of the multitenant software infrastructure can easily create and manage a physically materialized schema version for the application's core. The individual schema versions of the tenants can be easily created virtually using BiDEL evolution scripts, which provides the whole expressiveness of the relational algebra and is not limited to additive SMOs. All columns in the individual schema versions are explicitly named and typed overcoming the major drawback of traditional multitenancy solutions.

The distinctions between shared and private data can also be realized with InVerDa by starting the evolution to any customer schema by partitioning the tables according to the customer's identity column. So far, the PARTITION SMO is merely a general solution ignoring the special characteristics and the optimization potential of supporting shared and private data directly. We leave the specialization of InVerDa to the well-understood characteristics of multitenancy applications for future work.

Future Work: The SMO-based evolution in InVerDa facilitates easy and robust evolution support for multitenancy applications and it is up to future work to conduct case studies and provide comfortable tool support. Further, the native support of the distinction between shared and private data is very special to the multitenancy scenario. Of course, InVerDa allows to add a column to each table holding the tenants' identifier, however, this does not consider the special characteristics and potentials for performance optimization of this tenant identifier column. However, this research question is orthogonal to InVerDa and can be answered stand-alone first.

8.3 PRODUCTION-TEST SYSTEMS AND ROLLING UPGRADES

Scenario with State of the Art: Having multiple co-existing schema versions within one database is not exclusively a requirement posed by users, but also developers need to have multiple schema versions accessible during development and deployment, e.g. for production-test systems or for rolling upgrades.

In **production-test systems**, the current version of the application continues running, while the developers can run and test the next version of the application in parallel in their own branch. At the database layer, both the current and the new schema version need to be fully accessible by the applications just like a common single-schema database. Thereby, data from the production system is also visible in the test system, which allows developers together with their customers to test and experience the new version with the actual data. Traditionally, developers would either copy the database to a test instance or they would create the new schema version virtually with views. Both cases imply significant development overhead. Copying the whole database takes a lot of time for large data sets and the replica is out of sync which becomes crucial if customers are involved in the testing. For very large databases, not only the data needs to be replicated but also the infrastructure, which is not acceptable in most cases. This calls for creating the new schema version only virtually with the help of views. Developers and testers need the new schema version to be fully updatable to adequately test the new application, however, these changes should not be visible in the production data. All these features currently need to be implemented manually, which causes a huge development overhead and is expensive and error-prone in practice [66].

Another common scenario is the **rolling upgrade** of a new version: A new version is only available for a chosen subset of the customers, while the majority continues using the previous version. Those customers that already use the new version are beta testers to validate the new version of the application under realistic conditions. Implementing such a rolling upgrade is done by manually implementing the delta code e.g. in the application itself or with views in the database [100, 105]. Views need to virtually provide the new schema version for the beta-testers and any updates need to be propagated back to the materialized previous schema version—as discussed before, the view update problem is hard to solve and requires significant manual implementation effort. When the remaining users are moved to the new schema version as well, all the data from physical tables of the old schema version needs to be evolved to match the new schema version, which is then accessible for all user as a regular database. Again, this physical migration is very expensive and error-prone. According to a survey in 2011, data migration problems such as production-test systems and rolling upgrades account for 31 % of the budget in software development projects [66]. InVerDa can significantly reduce this effort and provide faster and more stable solutions.

Scenario with InVerDa: Realizing a production-test system with InVerDa is truly easy. Developers merely create a new schema version as a hot or cold copy depending on whether updates in the production schema version should be visible in the test schema version as well. InVerDa automatically takes care of providing both schema versions as intended.

For a rolling upgrade to a newly developed version, developers can also use InVerDa to let both schema versions co-exist at the same time, so both normal users and beta-testers can access their respective schema version, but any written data is immediately visible in both schema versions for all of them. As soon as all users moved to the new schema version, the developers can instruct InVerDa to physically move the data to the new schema version with the click of button and finally remove the old schema version from the database. InVerDa greatly supports developers in managing the development and deployment of new application versions with evolving database schemas. InVerDa easily facilitates robust production-test systems as well as rolling upgrades, releasing developers from the error-prone manual implementation.

Future Work: A major aspect of production-test systems that is not covered by InVerDa yet, is the efficient data management for hot and cold copies. A promising direction of future work is to not create hot and cold copies by initially copying the whole data at evolution time but by storing only the delta. Creating a test schema version with InVerDa that gets updates from the production database schema but does not propagate any write operation back, currently requires to fully replicate the data for the test schema version. A more appealing solution is to create the test schema version only virtually as a non-materialized schema version with InVerDa; however, the delta of any write operation performed on the test schema version must be stored and without propagating it back to the materialized production schema version. Hence, the promising research question is, how to store only the delta of this write operation in order to obtain a fast but also correct query processing on both schema versions.

8.4 THIRD-PARTY COMPONENT CO-EVOLUTION

Scenario with State of the Art: Modern software systems are composed of multiple components each one encapsulating e.g. certain features or services. Often these components are not all developed by the same stakeholders; assume a big enterprise solution like the SAP business suite, where SAP provides merely the core application and a whole industry of third-party component developers is settled around SAP extending the SAP core e.g. for different business fields. These third-party components extend and modify the behavior of the actual core application. Whenever the core application is further developed to the next version and rolled out to the customer, all the third-party components require a review as well—otherwise we run into the risk to end up with non-working features, inconsistent data or complete crashes. Particularly, the core might apply changes to the database schema—when the new schema differs from the schema the third-party extension is expecting, queries cannot be executed correctly and we end up with non-working applications or inconsistent data [66].

Currently, developers co-evolve the extensions manually for every evolution performed on the core's database schema, which is an error-prone and expensive task in practice. There is no reasonable tool support which we blame to the fact that database evolution is not specified with proper DELs these days but with traditional SQL or even within application code. This makes it notoriously hard to extract the essence and the intention of a variant or core evolution in the first place. Knowing the developers' intentions would be the most important ingredient to apply e.g. the intention of the variant evolution to the evolved core.

Scenario with InVerDa: With InVerDa's miX (Chapter 5), developers get great support in semi-automatically co-evolving all third-party components with the application's core. Given that both the evolution of the core and the evolution of the third-party components are written as sequences of BiDEL SMOs, we can easily merge the third-party component into the new core, which returns a co-evolved schema version. This new database schema for the co-evolved third-party component still represents the intention of the third-party component but now based on the new version of the core. This might entail that queries of the third-party component cannot be executed any longer, e.g. when the core drops a table that is accessed by the third-party component. However, this is obviously intended by the core evolution and therefore needs to be reflected in the third-party component as well. Currently, miX cannot support this task, so we emphasize this as promising future work.

An alternative option facilitated by InVerDa is to continuously run the third-party component without adjusting its database schema at all. InVerDa is able to propagate any read or write access in the third-party component through the core evolution to the new evolved version of the core, which allows to run both the new core version and the old third-party component upon the same logical data set. Nevertheless, eventually the application logic of the third-party components should be co-evolved with the core to reduce the propagation overhead and to incorporate the core changes into the third-party component as well. Refactoring the application code of the third-party component to match the new schema version is currently not covered by InVerDa but definitively an interesting opportunity to dig into deeper.

Future Work: Even though we can already cover large parts of the third-party component co-evolution with InVerDa's miX, there are promising future research challenges to further simplify the process. Most importantly, queries written in the third-party components may become invalid after co-evolving the schema. In these cases, it would be desirable to have tool support that uses the intentions captured in BiDEL's SMOs (semi-) automate the query rewriting. Another interesting research challenge is to further increase BiDEL's expressiveness so that the SMOs cover solution strategies for resolving ambiguities and other conflicts. Currently, miX consults the developers to provide a solution strategy, however, providing general answers to the possible questions already at evolution time would allow to fully automate the third-party component co-evolution with miX.

8.5 ZERO DOWNTIME MIGRATION

Scenario with State of the Art: Upgrading a software system and especially the database from one version to the next is not only error-prone and expensive as intensively discussed in this thesis, but also very time consuming. Given the upgrade includes an evolution of the database schema with restructurings of the data, the whole currently existing data needs to be migrated to the new schema version. For common enterprise-sized database this can easily take hours or even days, making the database unavailable during the whole upgrade process. For most businesses it is simply not acceptable to have a downtime that temporarily stops the whole business.

The goal is to perform an upgrade including database migration with zero downtime. Currently, developers have to manually implement the zero downtime migration logic, which is again error-prone and expensive. The best practice is to keep the old schema version alive and create the new schema version hidden in the background. While silently copying the data step by step in the background, additional delta code on the old schema version propagates any updates in the old schema version to those tuples that have already been copied to the new schema version [100]. After successfully migrating all the data in the background from the old to the new schema version, the developers can make the new schema version available and remove the old one without moving any further data, which is then possible with almost no downtime.

Scenario with InVerDa: InVerDa decouples the physical migration from one schema version to another from their logical evolution; so we can postpone a physical migration while already using the new schema version until a time, when downtime is more acceptable. Zero downtime migration is currently not supported by InVerDa.

Future Work: BiDEL's SMOs carry enough information to generate all the delta code for the discussed manual solution automatically, which is a great potential that should be discovered in future work. Data can be copied stepwise in the background following the SMOs' semantics and subsequent changes on the old schema version can easily be propagated to those tuples already copied to the new schema version without any further interaction of the developers. There are several challenging research questions to realize zero downtime migration based on BiDEL. Most interestingly, we have to find the right granularity for the stepwise migration in the background: There is a continuum ranging from tuple-per-tuple to table-per-table. While the extremes cause either too much overhead for copying every single tuple separately or imply downtime for locking a whole table, there might be better—even adaptive—solutions in between these extremes. Thus, there is a trade-off between the overall migration time and the performance reduction during the migration, which needs to be carefully discovered and analyzed. Another interesting research questions addresses the guaranteed correctness that the data at the new schema version is in the end equivalent to locking the whole database and migrating the data in one haul, especially when the schema versions are updated during the physical migration. The currently applied manual implementation of zero downtime migrations is error prone and expensive. In sum, InVerDa facilitates promising future research to fully automate zero-downtime migration for BiDEL specified evolution since the SMOs carry all the information to generate the traditional zero-downtime migration automatically.

8.6 EXTENSIONS OF BIDEL

Scenario with State of the Art: Database evolution is not exclusively limited to the evolution of tables and the data stored in those tables. During the software development process all artifacts in a database can be evolved eventually. Beyond the tables, this covers e.g. constraints, views, physical data structures like indices or materialized views, and especially functional database objects like stored procedures, user defined functions, or triggers. The evolution of most of these artifacts has already been proposed as database refactorings [8], however, this does only cover the plain notion that these artifacts can be changed—an evolution language similar to BiDEL is missing but necessary to automate further database evolution tasks like the merging of schema versions or the propagation of data accesses between co-existing schema versions also for the other artifacts in a database.

Considering e.g. the evolution of physical database objects like materialized views: A schema version realized with the help of updatable views should be materialized in order to speed up query processing. However, the delta code (view definitions) for all subsequent schema versions need to be updated manually now as well, since they can access the materialized view as well. Further, the evolution of functional database artifacts, like evolving the behavior of stored procedures or simply adapting them to evolving base tables, poses tough challenges on database developers. The currently applied manual solutions are very expensive and error-prone in practice.

Scenario with InVerDa: InVerDa is in fact limited to the evolution of tables in a relational database. We have shown that BiDEL is relationally complete; any evolution of the database schema that can be expressed with relational algebra expressions can be identically achieved with a sequence of BiDEL SMOs. The evolution of constraints and views, as well as physical and functional database artifacts is out of scope for now. To make InVerDa practically appealing, it is important future research to extend the ideas of InVerDa to the evolution of the remaining artifacts in a relational database.

Future Work: In the following, we discuss the potential and challenges for future research addressing the evolution of the most important database artifacts that are currently not covered by BiDEL. The evolution of **constraints** is already partly covered by PRISM++ [41]; however, the support for merging schema versions and to let them co-exist is still missing and promising future work. For the evolution of **views**, there are basically two options: we either preserve the result of the view by inverting the evolution of the tables, which is easily possible with InVerDa, or we co-evolve the view with the base table evolution. The latter is equal to merging schema versions, as long as the view is specified with BiDEL—otherwise the co-evolution of views is a hard research problem since the intention of the view is unclear.

The evolution of **physical database artifacts** like indices, materialized views, etc. has been out of scope because in an ideal world they should not be visible to the database developers anyhow. With InVerDa, we intentionally strengthened the separation between the logical database schema versions and the physical representation to achieve full data independence. This raises the question to which extent the DBA should determine the physical database schema—we see three different extents of physical database evolution, which should be analyzed in future research. First, the DBA is not involved in the physical database design at all but relies on a good adviser that is not limited to choosing the set of materialized table versions but also creates further physical database artifacts as needed for a high performance. Second, the DBA can give hints to the adviser that are incorporated to the physical database design as long as they make sense; e.g. indices for table versions that are not materialized are not possible. Third, the DBA is equipped with a manual editor that provides tool support for reliable predictions on performance measures and supports the co-evolution of all other affected artifacts in the database.

Till now, we only evolved structures but no behavior. In order to evolve **functional database artifacts** like stored procedures and triggers as well, we first need to rethink the way of defining them in the first place. Similar to BiDEL for tables, we need an evolution language that captures the intention of the functions and allows to co-evolve them with the base tables as well as evolving the functions itself in a controlled and robust manner. To achieve this goal, there are plenty of interesting research questions that need to be answered covering different areas like (1) the definition of a feasible evolution language, (2) the actual evolution of functions and their co-evolution with depending database artifacts, (3) testing, and (4) formal guarantees regarding the correct execution of evolved functions.

In sum, InVerDa currently focuses on the expressiveness of the relational algebra which covers the evolution of tables—the primary database artifacts that actually hold the payload data. However, in order to become practically applicable, InVerDa also needs to cover all other database artifacts that are well-established in current database systems, which motivates promising future work.

8.7 EFFICIENT SYNCHRONIZATION OF SCHEMA VERSIONS

Scenario with State of the Art: Keeping multiple different schema versions in sync is not only a matter of correct information preserving propagation and persistence of the data but also a matter of performance. The overhead for the transformation of data accesses between schema versions tends to grow with the complexity of the evolution between two schema versions. The current practical solution is to hand-tune the written delta code that propagates data accesses between schema versions until the performance becomes acceptable. This manual approach is very time consuming and expensive since manual performance optimizations usually involve to deep-dive from the level of relational tables down to indices, pages, and records. Further, hand-tuned delta code is also more error-prone as the code's clarity and maintainability suffers from the technical deep-dive to optimize the delta code.

Scenario with InVerDa: For reading or writing data at any schema version, InVerDa propagates the data access through the evolution's SMOs to the materialized table versions. Thereby, auxiliary tables represent the information that needs to be stored additionally in order to prevent it from being lost. We currently map these auxiliary tables from the Datalog rules *1:1* to physical tables in the database, however, this database design is most likely far from optimal. Further, the current implementation propagates data access stepwise through each SMO using cascading views for reading and cascading triggers for writing. As there is one view for each intermediate table version, the propagation logic covers one SMO at a time, which is great to ensure correctness and yields a robust and maintainable solution for InVerDa. However, the view mechanism disturbs the database optimizer's work: We have seen execution plans in PostgreSQL with unused optimization potential as selection predicated were not pushed down through cascading views, which results in a significant and truly unnecessary performance overhead. The same applies to the cascading trigger calls.

Future Work: Promising future work aiming at an increased performance for data access propagation should target two major aspects. First, the design of the auxiliary tables is crucial to provide the auxiliary information in a structure easily digestible by the databases' query processor. Second, the composition of SMO can be used to reduce the number of cascading views or triggers. Let us discuss these two opportunities in detail.

The set of Datalog rules specifying the semantics of the SMOs' mappings represents the auxiliary information in a normalized format—keeping the normalized design for the auxiliary tables can imply unnecessary performance overheads. Assuming an evolution with N subsequent `ADD COLUMN` SMOs, InVerDa creates N auxiliary tables holding the values of the respectively added column, which results in N expensive joins for reading the data. An **optimized design of the auxiliary tables** could prejoin all the auxiliary tables reducing the overhead to one single join.

In general, the structure of auxiliary tables and their usage during query processing is not very diverse, which makes a general optimization solution for the design of auxiliary tables a promising future work. We identify two major types of auxiliary information. First, auxiliary information can simply be a list of tuple identifiers p used to flag and filter tuples of the data tables. The number of joins can be reduced significantly, by denormalizing the auxiliary tables and prejoining the auxiliary information of one or multiple SMOs either with each other or with the actual data tables. Second, other auxiliary information are sets of tuples with the same attributes as a data table, e.g. tuples that are not matched by the partitioning criteria. One promising physical table design is to append the auxiliary tuples to the actual data table and mask them with a flag to be excluded for reading the plain data table. All alternative physical database designs should be defined, structured, and evaluated in future work. When designing auxiliary tables, special focus can be laid on choosing beneficial physical structures for the different types of auxiliary information, like e.g. specialized indices to store and retrieve the data even more efficiently.

The other research opportunity to increase data access performance is the **composition of the SMOs' mapping semantics** for both reading and writing. A promising research opportunity is to avoid the chaining of views and triggers and let the views/triggers immediately access the data table and auxiliary table even when they are multiple SMOs away along the schema version history. This requires a representation of the propagation logic between schema versions that can be easily composed and simplified to generate efficient SQL view and trigger definitions for the composed data access propagation. Datalog proved to be a great formalism for the formal evaluation of the SMOs' bidirectionality as well as for the generation of views/triggers for the stepwise propagation. However, for combining and reducing rules in an automated fashion, it turned out that Datalog rules are too far away from the actual query processing in a relational database and make many optimizations inadequately hard. E.g. an outer join is expressed with three different Datalog rules for (1) matching tuples, (2) orphan left tuples, and (3) orphan right tuples. When optimizing such a rule set or generating delta code

from such a rule set, this outer join is not explicitly visible and will be treated as a union of three separate select statements, which is not efficient. Therefore, a promising idea for future research is to use the relational algebra to represent the data propagation of each single SMO. As relational algebra expressions are trees, they can be easily plugged together and there is an enormous body of literature regarding their logical optimization. Generating the views and triggers directly from these optimized relational execution plans removes InVerDa's cascading execution and should increase the performance significantly.

As a conclusion, the major research challenges are (1) the physical design of auxiliary tables for the auxiliary information both of a single SMO and for sequences of SMOs and (2) the composition and optimization of SMO mappings to shorten the distance between a schema version and the materialized data tables. Obviously, these two challenges influence each other: The query processing through multiple SMOs can be further simplified when auxiliary tables are e.g. already prejoint. Similarly, the composition of auxiliary tables across SMOs just makes sense when the data propagation of subsequent SMOs is executed together in one shot.

CONCLUSION

We started out with the goal to make database development just as agile as software development. We will now look back and conclude this thesis by summarizing our contributions to achieve this goal. Agile database development calls for an easy and robust schema evolution in a production database—however, this is not as easy as evolving application code since the database already stores data in the initial schema version and we need to evolve the existing data accordingly. Evolving a production database without compromising the data's correctness and completeness is a tough challenge in practice that accounts for significant costs in software projects. Further, build systems allow to deploy and run multiple versions of an application concurrently e.g. to provide different versions for differently progressive users, to run test and production systems concurrently, or to let multiple different subsystems of software system access the same database with different schema versions. The same is hard at the database end. The database as single point of truth cannot easily provide co-existing schema versions as such support is missing in current DBMSes. Instead, developers end up writing delta code by hand to e.g. provide co-existing schema versions with data access propagation between them or to physically migrate the database without limiting the availability of any schema version, which is very expensive and error-prone in practice.

To this end, we proposed **multi-schema-version database systems** that allow to have co-existing schema versions within a single database. New schema versions can be created very easily by e.g. evolving an existing schema version with an intuitive DEL or by merging two existing schema versions to a new schema version. Newly created schema versions are immediately available and can be read and written just like any other schema version. Multi-schema-version database systems provide full data independence, hence the DBA can freely move or replicate the materialization along the schema versions history without limiting the availability of any schema version. In Chapter 2, we conducted a detailed analysis of the requirements in multi-schema-version database systems and derived an architecture that realizes multi-schema-version data management upon any common relational DBMS. Our proposed system is called InVerDa and generates all the delta code for co-existing schema versions with full data independence automatically. InVerDa generates views for each table version in each schema version and makes them updatable using instead-of triggers on those views. Thanks to this architecture, InVerDa is only active during development and migration time—at runtime, the applications access the data using standard DBMS technology. Thereby, we ensure that we do not lose the performance, the transaction guarantees, or any other well-established feature of relational databases. Realizing multi-schema-version database systems comes with a wide bouquet of research questions that touch both very old fields such as the view update problem and materialized view maintenance but also more recent topics like SMO-based database evolution. In Chapter 3, we conducted a detailed literature review and showed that there are many important approaches tackling parts of multi-schema-version database systems, but also that we are the first to provide truly co-existing schema versions with full data independence.

To describe the evolution from an existing schema version to a new one, we introduced a DEL, called BiDEL in Chapter 4. BiDEL couples the evolution of both the schema and the data in compact and intuitive SMOs. BiDEL is a **relational complete DEL**: Every evolution that can be described with relational algebra expressions can be expressed with BiDEL as well. We have formally evaluated the relational completeness and we have also shown that BiDEL is applicable for realistic scenarios such as the evolution of 171 schema versions of Wikimedia, which makes BiDEL a great basis for our multi-schema-version database system InVerDa. Further, we defined additional parameters for BiDEL's SMOs to make them **bidirectional**. Those parameters need to be provided by the developers if schema versions should co-exist, which requires bidirectional propagation of data accesses between the co-existing schema versions. The bidirectional SMOs allow to generate delta code for co-existing schema versions automatically without any further interaction of any developer—according to our empirical evaluation, the developers write orders of magnitude less code when they create new schema versions with BiDEL instead of traditional SQL.

As a second way to create new schema versions in a multi-schema-version database system, we introduced **MIX** in Chapter 5. MIX is a semi-automatic consolidation algorithm that **merges two existing schema versions** into a new schema version. The developers provide two schema versions that have been developed independently and originate from a common source schema—please note that this common source schema can be the empty one, which makes merging trivial. The two BIDEL-specified evolutions from the common source schema version to the respective schema versions precisely capture the developers' intentions, which can then be merged automatically. During the merging process, MIX basically consolidates every pair of SMOs from the two evolutions stepwise—when manually merging two schema versions, the developers would have to do the same, which sums up to $N \times M$ combinations with N and M being the lengths of the two evolutions. Hence, MIX reduces the effort for the developers to a minimum: The proposed consolidation algorithm MIX is semi-automatically and merely asks the developers very simple questions in case the two evolutions conflict, e.g. to resolve naming conflicts when both evolution create table versions or columns with the same name. In sum, we presented two means to create new schema versions in a multi-schema-version database system, namely the evolution of one existing schema version with BIDEL or the semi-automatic merging of two existing schema versions with MIX.

All the created schema versions co-exist in the multi-schema-version database system and can be accessed by applications just as any common relational database. If full bidirectional propagation of data between schema versions is enabled, than all the co-existing schema versions provide different views on the same conceptual data set. Naïvely, we could materialize every created table version physically and merely propagate the write operations between them; this fully redundant materialization yields the highest read performance since there is no need to rewrite any query. However, write operations face a high overhead for updating all affected table versions and a lot of space is wasted since all schema versions basically show the same conceptual data set, which is then stored highly redundant. As a solution, the bidirectionality of BIDEL's SMOs allows to non-redundantly materialize merely a slice of the schema versions catalog—since not all SMOs are information preserving, we have to manage the otherwise lost information in auxiliary tables. In Chapter 6, we defined the **data independent** semantics of BIDEL's SMOs: Depending on which side of an SMO we actually materialize the data, different auxiliary information need to be managed—either way, the table versions at each side of the SMO behave like tables in a common relational database. We formally showed that even the non-redundant materialization provides co-existing schema versions, where each schema version behaves like a regular single-schema database. This particularly requires two things: First, data written in a table version that is not materialized is propagated through SMOs to materialized table versions—when reading it again, the initially written data can be read completely and unchanged. Second, a write operation in one table version of a schema version has no side effects on other table versions of the same schema version, since this would contradict the developers' assumptions in common relational databases. By formally showing that these characteristics hold and that data can be materialized independently of the co-existing schema versions in a subset of the table versions, we can now guarantee full data independence. This allows using a multi-schema-version database system without the risk of losing the transaction guarantees of relational databases.

We have shown the data independence mainly for the corner case of non-redundant materialization. We can easily derive the data management of co-existing schema versions with partially redundant materializations: For all SMOs that have both the source and the target side materialized, we simply leave out the propagation to auxiliary tables since all the relevant information is stored at either side of the SMO. For all table versions that are not materialized, we retrieve the data as in the non-redundant case from the cost-wise closest materialized table versions. This way, we opened up a huge space of possible materializations and thanks to the formally guaranteed data independence, we know that any of the possible materializations provides all co-existing schema versions without any information being lost and without any unexpected side effects.

The only effect of changing the materialization is a changing performance. According to our evaluation, we can execute read and write operations orders of magnitude faster by choosing the right materialization w.r.t. the current workload. In Chapter 7, we analyzed the detailed characteristics of the overhead for propagating data accesses through BiDEL's SMOs. The overhead for propagating data accesses through SMOs differs both with the type of the SMO and with the direction of propagation. The involved auxiliary tables have a major influence on the actual performance overhead. Unfortunately, it is very difficult for DBAs to fully grasp the details of the technical implementation and the resulting effects on the performance. Especially since the search space of possible materializations can grow exponentially with the number of table versions, there is no feasible way for a DBA to reliably determine the best materialization whenever the workload changes. Therefore, we introduced a **cost model-based adviser** that learns a cost model of the multi-schema-version database system and proposes a materialization that is optimized for the current workload and stays within given memory boundaries. In the evaluation, it has proven to significantly improve the performance of changing workloads by proposing partially redundant materializations that are not trivial to find for DBAs.

The presented multi-schema-version database systems are very helpful for agile database development, but there are way more scenarios in practice where the presented concepts can be applied as we discussed in Chapter 8. Specifically, we discussed how to apply multi-schema-version database systems in software product line engineering, in multitenancy application development, in production-test scenarios, as well as for third-party component co-evolution. These scenarios are already fully realizable with InVerDa or require merely minor extensions. Further, we discussed future research questions to extend multi-schema-version database systems in general to cover even more scenarios. For instance, the concept of temporarily co-existing schema versions can be used to implement zero-downtime migration to new schema versions. Further, the expressiveness of BiDEL is currently limited to the expressiveness of the relational algebra, but it should be extended to further non-functional and functional database artifacts in order to cover all practically relevant scenarios. Finally, the propagation of data accesses between schema versions has been considered on a logical level to formally evaluate the correctness of the data independent data management so far—there is a huge research potential to speed up data access propagation by fine-tuning the management of auxiliary information and by combining and reducing the data access propagation through chains of subsequent SMOs.

In sum, multi-schema-version database systems appear to be a useful tool for agile database development, as they allow to easily create new schema versions by evolving or merging existing ones; these new schema versions can co-exist with all other schema versions. Thanks to the formally evaluated data independence, we can use the presented adviser to find and apply the best materialization for the given workload without the risk of losing any data. No matter which materialization we chose, all schema versions are guaranteed to behave like common single-schema databases. The presented multi-schema-version database systems can be applied in many practically relevant scenarios and open up a wide field of promising future research.

BIBLIOGRAPHY

[1] Interim Report: ANSI/X3/SPARC Study Group on Data Base Management Systems 75-02-08. *FDT - Bulletin of ACM SIGMOD*, 7(2):1–140, 1975.

[2] Lamia Abo Zaid and Olga De Troyer. Towards Modeling Data Variability in Software Product Lines. In *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing (LNBIP)*, pages 453–467. Springer, 2011.

[3] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. Feature Assembly Modelling: A New Technique for Modelling Variable Software. In *International Conference on Software and Data Technologies (ICSOFT)*, pages 29–35. SciTePress, 2010.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 496–505. VLDB Endowment, 2000.

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Materialized view and index selection tool for Microsoft SQL server 2000. *ACM SIGMOD Record*, 30(2):608, 2001.

[6] Suad Alagic and Philip A. Bernstein. A Model Theory for Generic Schema Management. In *International Workshop on Database Programming Languages (DBPL)*, pages 228–246. Springer, 2001.

[7] Scott W. Ambler and Mark Lines. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012.

[8] Scott W. Ambler and Pramod J. Sadalage. *Refactoring databases: Evolutionary database design*, volume 4. Addison-Wesley Signature, 2006.

[9] Chris J. Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 2010.

[10] Meenakshi Arora and Anjana Gosain. Schema Evolution for Data Warehouse: A Survey. *International Journal of Computer Applications (IJCA)*, 22(5):6–14, 2011.

[11] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *International Conference on Management Of Data (SIGMOD)*, pages 1195–1206. ACM, 2008.

[12] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. A comparison of flexible schemas for software as a service. In *International Conference on Management Of Data (SIGMOD)*, pages 881–888. ACM, 2009.

[13] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and Data Sharing in evolving multi-tenant databases. In *International Conference on Data Engineering (ICDE)*, pages 99–110. IEEE, 2011.

[14] David E. Avison and Guy Fitzgerald. Where now for development methodologies? *Communications of the ACM*, 46(1):78–82, 2003.

[15] François M. Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.

[16] George M. Beal and Joe M. Bohlen. *The diffusion process*. Agricultural Experiment Station, Iowa State College, 1957.

[17] Kent Beck. *Extreme Programming. Das Manifest*. Addison-Wesley, 2000.

[18] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. *http://agilemanifesto.org/*, 2001.

[19] Andreas Behrend, Ulrike Griefahn, Hannes Voigt, and Philip Schmiegelt. Optimizing continuous queries using update propagation with varying granularities. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–12. ACM, 2015.

[20] Mathieu Beine, Nicolas Hames, Jens H Weber, and Anthony Cleve. Bidirectional Transformations in Database Evolution: A Case Study "At Scale". In *CEUR Workshop*, pages 100–107. CEUR-WS.org, 2014.

[21] Herbert D. Benington. Production of Large Computer Programs. *IEEE Annals of the History of Computing*, 5(4):350–361, 1983.

[22] Peter Bentley and David Corne. *Creative evolutionary systems*. Morgan Kaufmann, 2002.

[23] Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing mapping composition. *The VLDB Journal*, 17(2):333–353, 2007.

[24] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic Schema Matching, Ten Years Later. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):695–701, 2011.

[25] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *International Conference on Management Of Data (SIGMOD)*, pages 1–12. ACM, 2007.

[26] Aaron Bohannon, Benjamin Pierce, and Jeffrey A. Vaughan. Relational lenses - A Language for Updatable Views. In *Symposium on Principles of Database Systems (PODS)*, pages 338–347. ACM, 2006.

[27] Michael L. Brodie and Jason T. Liu. Keynote: The Power and Limits of Relational Technology In the Age of Information Ecosystems. In *OnTheMove Federated Conferences & Workshops (OTM)*, pages 2–3, 2010.

[28] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal Data Partitioning in Database Design. In *International Conference on Management Of Data (SIGMOD)*, pages 128–136. ACM, 1982.

[29] 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 Transactions on Computer Systems (TOCS)*, 26(2):4:1–4:26, 2008.

[30] Hai-Tao Chen, Hu-Sheng Liao, Hai-Tao Chen, and Hu-Sheng Liao. A Survey to View Update Problem. *Journal of Electronic Science and Technology (JEST)*, 8(4):318–327, 2011.

[31] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *International Conference on Requirements Engineering (RE)*, pages 31–40. IEEE, 2005.

[32] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[33] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *International Conference on Enterprise Distributed Object Computing (EDOC)*, pages 222–231. IEEE, 2008.

[34] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in Database Applications Evolution. In *International Conference on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 409–421. Springer, 2005.

[35] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, pages 377–387, 1970.

[36] Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of Relational Views. *Journal of the ACM*, 31(4):742–760, 1984.

[37] Carlo Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, 2013.

[38] Carlo Curino, Hyun J. Moon, Myungwon Ham, and Carlo Zaniolo. The PRISM Workwench: Database Schema Evolution without Tears. In *International Conference on Data Engineering (ICDE)*, pages 1523–1526. IEEE, 2009.

[39] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):761–772, 2008.

[40] Carlo Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 323–332. Springer, 2008.

[41] Carlo Curino and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):117–128, 2010.

[42] Christopher J. Date. *View updating and relational theory*. O'Reilly Media, Inc., 2012.

[43] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *International Conference on Very Large Data Bases (VLDB)*, pages 368–377. VLDB Endowment, 1978.

[44] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo – Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205, 2007.

[45] Eladio Domínguez, Jorge Lloret, Ángel L. Rubio, and María A. Zapata. MeDEA: A database evolution architecture with traceability. *Data and Knowledge Engineering (DKE)*, 65(3):419–441, 2008.

[46] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing schema mappings. *ACM Transactions on Database Systems*, 30(4):994–1055, 2005.

[47] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Quasi-inverses of schema mappings. *ACM Transactions on Database Systems*, 33(2):1–52, 2008.

[48] Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. *Empirical Studies of Open Source Evolution*, pages 263–288. Springer, 2008.

[49] C. A. Floudas, P. M. Pardalos, C. Adjiman, W. R. Esposito, Z. H. Gümüs, S. T. Harding, J. L. Klepeis, C. A. Meyer, and C. A. Schweiger. *Handbook of Test Problems in Local and Global Optimization*. Nonconvex Optimization and Its Applications. Springer, 2013.

[50] Nathan J. Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[51] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[52] Enrico Franconi and Paolo Guagliardo. The View Update Problem Revisited. *CoRR*, abs/1211.3, 2012.

[53] Pierre Genevès, Nabil Layaïda, and Vincent Quint. Impact of XML Schema Evolution. *ACM Transactions on Internet Technology (TOIT)*, 11(1):1–27, 2011.

[54] Matteo Golfarelli, Jens Lechtenbörger, Stefano Rizzi, and Gottfried Vossen. Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. *Data & Knowledge Engineering (DKE)*, 59(2):435–459, 2006.

[55] Fabio Grandi. A Relational Multi-Schema Data Model and Query Language for Full Support of Schema Versioning. In *Italian Symposium on Database Systems (SEBD)*, pages 323–336, 2002.

[56] Ashish Gupta and Inderpal Singh. Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.

[57] Michael Hartung, James F. Terwilliger, and Erhard Rahm. Recent Advances in Schema and Ontology Evolution. In *Schema Matching and Mapping*, pages 149–190. Springer, 2011.

[58] Kai Herrmann, Jan Reimann, Hannes Voigt, Birgit Demuth, Stefan Fromm, Robert Stelzmann, and Wolfgang Lehner. Database Evolution for Software Product Lines. In *International Conference on Data Management Technologies and Applications (DATA)*, pages 125–133. SciTePress, 2015.

[59] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. CoDEL – A Relationally Complete Language for Database Evolution. In *European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 63–76. Springer, 2015.

[60] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *International Conference on Management Of Data (SIGMOD)*, pages 1101–1116. ACM, 2017.

[61] Kai Herrmann, Hannes Voigt, Jonas Rausch, Andreas Behrend, and Wolfgang Lehner. Robust and simple database evolution. *Information Systems Frontiers*, (Special issue):1–17, 2017.

[62] Kai Herrmann, Hannes Voigt, Thorsten Seyschab, and Wolfgang Lehner. InVerDa – co-existing schema versions made foolproof. In *International Conference on Data Engineering (ICDE)*, pages 1362–1365. IEEE, 2016.

[63] Kai Herrmann, Hannes Voigt, Thorsten Seyschab, and Wolfgang Lehner. InVerDa – The Liquid Database. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 619–622. Gesellschaft für Informatik (GI), 2017.

[64] Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution: A transformational approach. *Data & Knowledge Engineering (DKE)*, 59(3):534–558, 2006.

[65] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. *Principles of Programming Languages (POPL)*, 46(1):371–384, 2011.

[66] Philip Howard. Data Migration Report, Bloor, 2011.

[67] Zhenjiang Hu, Andy Schurr, Perdita Stevens, and James F. Terwilliger. Dagstuhl seminar on bidirectional transformations (BX). *ACM SIGMOD Record*, 40(1):35, 2011.

[68] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *International Conference on Management Of Data (SIGMOD)*, pages 1259–1274. ACM, 2017.

[69] Google Inc. Google Datastore. *https://developers.google.com/appengine/docs/java/datastore/*, 2013.

[70] Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt, and Wolfgang Lehner. Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 157–176. Gesellschaft für Informatik (GI), 2015.

[71] Tobias Jäkel, Thomas Kühn, Hannes Voigt, and Wolfgang Lehner. RSQL - A Query Language for Dynamic Data Types. In *International Database Engineering & Applications Symposium (IDEAS)*, pages 185–194. ACM, 2014.

[72] Christian Kaas, Torben Bach Pedersen, and Bjørn Rasmussen. Schema Evolution for Stars and Snowflakes. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 425–433. Springer, 2004.

[73] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and Spencer A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Pittsburgh: Software Engineering Institute, Carnegie Mellon University, 1990.

[74] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Symposium on Principles of Database Systems (PODS)*, pages 154–163. ACM, 1985.

[75] Niloofar Khedri and Ramtin Khosravi. Handling Database Schema Variability in Software Product Lines. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 331–338. IEEE, 2013.

[76] Meike Klettke. Conceptual XML Schema Evolution-the CoDEX Approach for Design and Redesign. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 53–63. Gesellschaft für Informatik (GI), 2007.

[77] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014.

[78] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. FRaMED: Full-fledge Role Modeling Editor (Tool Demo). In *International Conference on Software Language Engineering (SLE)*, pages 132–136. ACM, 2016.

[79] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering (SLE)*, pages 141–160. Springer, 2014.

[80] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. In *ACM SIGMOD Record*, volume 41, pages 34–43. ACM, 2012.

[81] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010.

[82] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[83] Max Leuthäuser and Uwe Aßmann. Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming. In *Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering (MORSE/VAO)*, pages 25–33. ACM, 2015.

[84] Harold Lim, Yuzhang Han, and Shivnath Babu. How to Fit when No One Size Fits. In *Conference on Innovative Data Systems Research (CIDR)*, pages 35–46, 2013.

[85] Jixue Liu, Chengfei Liu, Theo Haerder, and Jeffery Xu Yu. Updating Typical XML Views. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 126–140. Springer, 2012.

[86] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. In *International Conference on Very Large Data Bases (VLDB)*, pages 49–58. Morgan Kaufmann, 2001.

[87] Petros Manousis, Panos Vassiliadis, Apostolos Zarras, and George Papastefanatos. *Schema Evolution for Databases and Data Warehouses*, pages 1–31. Springer, 2016.

[88] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. In *International Workshop on Multi-Paradigm Modeling (MPM)*, volume 42, pages 1–13. EASST, 2012.

[89] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *International Conference on Management Of Data (SIGMOD)*, pages 307–318. ACM, 2001.

[90] Hyun J. Moon, Carlo Curino, Myungwon Ham, and Carlo Zaniolo. PRIMA – archiving and querying historical data with evolving schemas. In *International Conference on Management Of Data (SIGMOD)*, pages 1019–1022. ACM, 2009.

[91] Hyun J. Moon, Carlo Curino, and Carlo Zaniolo. Scalable architecture and query optimization fortransaction-time DBs with evolving schemas. In *International Conference on Management Of Data (SIGMOD)*, pages 207–218. ACM, 2010.

[92] Stephan Murer, Bruno Bonati, and Frank J Furrer. *Managed Evolution: A Strategy for Very Large Information Systems*. Springer, 2010.

[93] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering – Foundations, Principles and Techniques*. Springer, 2005.

[94] Li Qian, Kristen LeFevre, and H. V. Jagadish. CRIUS: user-friendly database design. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):81–92, 2010.

[95] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 125–135. ACM, 2013.

[96] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[97] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *ACM SIGMOD Record*, 35(4):30–31, 2006.

[98] John F. Roddick. Schema Evolution in Database Systems – An Annotated Bibliography. *ACM SIGMOD record*, 21(4):35–40, 1992.

[99] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[100] M. Ronström. On-line schema update for a telecom database. In *International Conference on Data Engineering (ICDE)*, pages 329–338. IEEE, 2000.

[101] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In *International Symposium on Database Programming Languages (DBPL)*. ACM, 2013.

[102] Stefanie Scherzinger, Uta Störl, and Meike Klettke. A Datalog-based protocol for lazy data migration in agile NoSQL Application development. In *International Symposium on Database Programming Languages (DBPL)*, pages 41–44. ACM, 2015.

[103] Pavel Shvaiko and Jérôme Euzenat. A Survey of Schema-Based Matching Approaches. In *Journal on Data Semantics IV*, pages 146–171. Springer, 2005.

[104] Ioannis Skoulis, Panos Vassiliadis, and Apostolos Zarras. Open-source databases: Within, outside, or beyond Lehman's laws of software evolution? In *International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 379–393. Springer, 2014.

[105] Daniel Sun, Len Bass, Alan Fekete, Vincent Gramoli, An Binh Tran, Sherry Xu, and Liming Zhu. Quantifying Failure Risk of Version Switch for Rolling Upgrade on Clouds. In *International Conference on Big Data and Cloud Computing (BDCloud)*, pages 175–182. IEEE, 2014.

[106] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. Consistent Unanticipated Adaptation for Context-Dependent Applications. In *International Workshop on Context-Oriented Programming (COP)*, pages 33–38. ACM, 2016.

[107] James F. Terwilliger. Bidirectional by Necessity: Data Persistence and Adaptability for Evolving Application Development. In *Summer School on Grand Timely Topics in Software Engineering (GTTSE)*, pages 219–270. Springer, 2011.

[108] James F. Terwilliger, Philip A. Bernstein, and Adi Unnithan. Automated co-evolution of conceptual models, physical databases, and mappings. In *International Conference on Conceptual Modeling (ER)*, pages 146–159. Springer, 2010.

[109] James F. Terwilliger, Philip A. Bernstein, and Adi Unnithan. Worry-free database upgrades: Automated Model-Driven Evolution of Schema and Complex Mappings. In *International Conference on Management Of Data (SIGMOD)*, pages 1191–1194. ACM, 2010.

[110] James F. Terwilliger, Anthony Cleve, and Carlo Curino. How Clean Is Your Sandbox? In *International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 1–23. Springer, 2012.

[111] James F. Terwilliger, Lois M. L. Delcambre, David Maier, Jeremy Steinhauer, and Scott Britell. Updatable and evolvable transforms for virtual databases. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1-2):309–319, 2010.

[112] Jeffrey D. Ullman. *Principles of database and knowledge-base systems*. Computer Science Press, 1988.

[113] Panos Vassiliadis, Apostolos V. Zarras, and Ioannis Skoulis. How is Life for a Table in an Evolving Relational Schema? Birth, Death and Everything in Between. In *International Conference on Conceptual Modeling (ER)*, pages 453–466. Springer, 2015.

[114] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 600–624. Springer, 2007.

[115] Bob Wall. Research on a Schema and Data Versioning System. In *VLDB PhD Workshop*, pages 43–48, 2011.

[116] Bob Wall and Rafal Angryk. Minimal data sets vs. synchronized data copies in a schema and data versioning system. In *Ph.D. Workshop on Information and Knowledge Management (PIKM)*, pages 67–73. ACM, 2011.

[117] Laurie Williams. Integrating pair programming into a software development process. In *Conference on Software Engineering Education and Training (CSEET)*, pages 27–36. IEEE, 2001.

[118] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 6(10):925–936, 2013.

[119] Yu Wu, Sushil Jajodia, and Sean X. Wang. Temporal database bibliography update. In *Temporal Databases: Research and Practice*, pages 338–366. Springer, 1998.

[120] Liqi Xu, Silu Huang, Sili Hui, Aaron J. Elmore, and Aditya Parameswaran. OrpheusDB: A Lightweight Approach to Relational Dataset Versioning. In *International Conference on Management Of Data (SIGMOD)*, pages 1655–1658. ACM, 2017.

[121] Yannakakis and Mihalis. Algorithms for acyclic database schemes. In *International Conference on Very Large Data Bases (VLDB)*, pages 82–94. VLDB Endowment, 1981.

[122] Liguo Yu and Alok Mishra. An empirical study of Lehman's law on software quality evolution. *International Journal of Software & Informatics (IJSI)*, 7(3):469–481, 2013.

[123] Carlo Zaniolo. Database relations with null values. In *Symposium on Principles of Database Systems (PODS)*, pages 27–33. ACM, 1984.

List of Tables

List of Figures

A

REMAINING BIDEL OPERATIONS

A.1 CREATE/RENAME/DROP TABLE AND RENAME COLUMN

The BiDEL operations for creating and dropping tables as well as renaming both tables and columns have trivial mappings and are implicitly bidirectional. For creating and dropping a table, there is only one reasonable materialization respectively: we materialize the `CREATE TABLE` SMO and virtualize the `DROP TABLE` SMO by default. So, we always materialize the schema version that actually holds the data table. As a counter example, a materialized `DROP COLUMN` SMO would need a single auxiliary table at the target side, which matches exactly the dropped data table from the source schema, so we can keep the SMO virtualized in the first place. Further, data accesses will never be propagated through an SMO that creates or drops a table, since data accesses can only be specified on the very side of the SMO, where the table really exists.

For renaming either tables or columns the mapping functions γ_{trg} and γ_{src} are also trivial, since no structure is changed. We merely rename named elements, which can be handled on the catalog level. The rewriting of any data access is straight forward: we access a renamed table by exchanging the table's name in the respective SQL statement and we rename a column using a simple projection in read operations or we rename the respective column in write operations.

A.2 ADD COLUMN & DROP COLUMN

SMO: `ADD COLUMN` b `AS` $f(r_1,\ldots,r_n)$ `INTO` R
Inverse: `DROP COLUMN` b `FROM` R `DEFAULT` $f(r_1,\ldots,r_n)$

We will focus the discussion on the `ADD COLUMN` SMO and formally evaluate its bidirectionality. Due to the bidirectional nature of the SMOs, we also implicitly evaluate the bidirectionality of the `DROP COLUMN` SMO. Adding a column is the only operation that allows calculating new values. It calculates the values of a new column b as a function over the existing values A and adds them to the table $R(p, A)$ to obtain the target version $R'(p, A, b)$. The following Datalog rule defines the calculation of the new table version R' in γ_{trg}.

$$R'(p, A, b) \leftarrow R(p, A), b = f_B(p, A) \tag{A.1}$$

The calculation of new values with the function f_B hits the limitation of traditional Datalog since termination can no longer be guaranteed. However, for our rule sets, we can still guarantee termination, because we do not have recursion at all. When storing this data in the source version, we project away the additional column b. Obviously, we need an additional auxiliary table B to store the values for the new column b since the function f does only provide initial values for b but both b and the columns A can be independently updated afterwards. We describe the γ_{src} mapping as discussed:

$$R(p, A) \leftarrow R'(p, A, \_) \tag{A.2}$$
$$B(p, b) \leftarrow R'(p, \_, b) \tag{A.3}$$

To complete the γ_{trg} mapping, we ensure that the values for the new column b are only calculated using the function f if no entry exists already in the auxiliary table B. So we end up with the following mapping functions for the `ADD COLUMN` SMO:

$\gamma_{\mathbf{trg}}$:

$$R'(p, A, b) \leftarrow R(p, A), b = f_B(p, A), \neg B(p, \_) \tag{A.4}$$
$$R'(p, A, b) \leftarrow R(p, A), B(p, b) \tag{A.5}$$

$\gamma_{\mathbf{src}}$:

$$R(p, A) \leftarrow R'(p, A, \_) \tag{A.6}$$
$$B(p, b) \leftarrow R'(p, \_, b) \tag{A.7}$$

Again we formally show that the bidirectionality criteria (Equation 6.1 and 6.2) are satisfied by calculating the resulting data after one round trip starting at source and target respectively.

A.2.1 $R_D = \gamma_{src}(\gamma_{trg}(R_D))$

$\gamma_{\mathbf{trg}}(\mathbf{R_D})$:

$$R'(p, A, b) \leftarrow R_D(p, A), b = f_B(p, A), \neg B(p, \_) \tag{A.8}$$
$$R'(p, A, b) \leftarrow R_D(p, A), B(p, b) \tag{A.9}$$

Initially, we consider all auxiliary tables to be empty: $B = \emptyset$. Hence, we can reduce the first rule and remove the second one according to Lemma 2.

$$R'(p, A, b) \leftarrow R_D(p, A), b = f_B(p, A) \tag{A.10}$$
$$\tag{A.11}$$

Now, we apply γ_{src} to this rule (Lemma 1).

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{R_D}))$:

$$R(p, A) \leftarrow R_D(p, A), b' = f_B(p, A) \tag{A.12}$$
$$B(p, b) \leftarrow R_D(p, A'), b = f_B(p, A') \tag{A.13}$$

Since the result of the calculation is never used in Rule A.12, we can remove the calculation from the rule's body according to Lemma 6.

$$R(p, A) \leftarrow R_D(p, A) \tag{A.14}$$
$$B(p, b) \leftarrow R_D(p, A'), b = f_B(p, A') \quad \square \tag{A.15}$$

Obviously, the data in R in the source version can be stored according to the target version without information loss. Reading it from there exactly returns the same tuples and the same values. Please note that we additionally store the generated values for the new column the auxiliary table B. Once a tuple has been inserted, the value of the new column is independent from the remaining columns, hence we have to store it explicitly.

A.2.2 R_D' = γ_{trg} (γ_{src} (R_D'))

To show that the second bidirectionality criteria also holds, we assume to have data R_D' at the target side, store it at the source side using γ_{src} and map it back to the target side with γ_{trg}.

$\gamma_{\mathbf{src}}(\mathbf{R'_D})$:

$$R(p, A) \leftarrow R'_D(p, A, \_) \tag{A.16}$$
$$B(p, b) \leftarrow R'_D(p, \_, b) \tag{A.17}$$

There are no auxiliary table at the rules' right side, so we cannot further reduce empty literals in these rules and therefore we directly apply γ_{trg} using Lemma 1.

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{R'_D}))$:

$$R'(p, A, b) \leftarrow R'_D(p, A, \_), b = f_B(p, A), \neg R'_D(p, \_, \_) \tag{A.18}$$
$$R'(p, A, b) \leftarrow R'_D(p, A, \_), R'_D(p, \_, b) \tag{A.19}$$

Rule A.18 can never be satisfied, as there are contradicting literals, so it can be removed (Lemma 4). Using the Lemma 5, we reduce Rule A.19: The common key p uniquely determines the values A and b, hence both literals can be extended to $R'_D(p, A, b)$ and obviously one of them can be eliminated using Lemma 7.

$$R'(p, A, b) \leftarrow \emptyset \tag{A.20}$$
$$R'(p, A, b) \leftarrow R'_D(p, A, b) \tag{A.21}$$

So, we finally end up with the reduced rule set for γ_{trg} (γ_{src} (R_D'):

$$R'(p, A, b) \leftarrow R'_D(p, A, b) \quad \square \tag{A.22}$$

In sum, we have shown that the `ADD COLUMN` SMO is bidirectional and the data survives any round trip completely and correctly. The auxiliary table B stores the values of the new column when the SMO is virtualized to ensure bidirectionality. With the projection to data tables, the SMO satisfies Conditions 6.1 and 6.2. For repeatable reads, the auxiliary table B is also needed when data is given in the source schema version.

	Decompose	Outer Join	Inner Join
ON PK	*A.4*	Inverse of A.4	*A.7*
ON FK	*A.5*	Inverse of A.5	Variant of A.8
ON Cond.	*A.6*	Inverse of A.6	*A.8*

Table A.1: Overview of different DECOMPOSE and JOIN SMOs.

A.3 DECOMPOSE & JOIN – GENERAL REMARKS

There are different extents for the DECOMPOSE and its inverse JOIN SMO: a join can have inner or outer semantics and it can be done based on the primary key, a foreign key, or on an arbitrary condition. As summarized in Table A.1, each configuration requires different mapping functions, however, some are merely the inverse or variants of others. The inverse of DECOMPOSE is OUTER JOIN and joining at a foreign key is merely a specific condition. The column sets of a DECOMPOSE SMO are not necessarily distinct and not necessarily covering the whole set of attributes of the source table. For those columns that occur in none of the resulting table versions we precede a drop column SMO and for columns occurring in both target table versions, we first duplicate the respective columns with an add column SMO to prevent from ambiguity.

To simplify the formal evaluation, we introduce two new lemmas that allow to apply the Lemma 1 in two separate steps. In a first step, Lemma 9 applies one rule set to another. In contrast to the previously used Lemma 1, we do not immediately reduce all rules but keep negated subequations explicitly, since this allows further reductions and shortens the whole evaluation significantly. Subsequently, we can use Lemma 10 to resolve the negated subexpressions. Please note that Lemma 1 is merely the sequential application of Lemma 9 and 10.

Lemma 9 (Application)**.** *Let $L \equiv q^r(p, Y)$ be a literal in the body of a rule r. For a rule $s \in \mathcal{R}^{pred(L)}$ let $rn(s, L)$ be rule s with all variables occurring in the head of s at positions of Y variables in L be renamed to match the corresponding Y variable and all other variables be renamed to anything not in $vars(body(r))$. If L is*

1. *a positive literal, s can be applied to r to get rule set $r(s)$ of the form $\{head(r) \leftarrow body(r) \setminus \{L\} \cup body(rn(s, L))\}$.*

2. *a negative literal, s can be applied to r to get rule set $r(s)$ of the form $\{head(r) \leftarrow body(r) \setminus \{L\} \cup \neg(body(rn(s, L)))\}$.*

For a given p, let r be every rule in $\mathcal{R}$ having a literal $L \equiv p(X, Y)$ in its body. Accordingly, $\mathcal{R}$ can be simplified by replacing all rules r and all $s \in \mathcal{R}^p$ with all $r(s)$ applications to $\mathcal{R} \setminus (\{r\} \cup \mathcal{R}^p) \cup (\bigcup_{s \in \mathcal{R}^{pred(L)}} r(s))$.

Lemma 10 (De Morgan's Law)**.** *Let $neg(r)$ be the set of all negated subexpressions (set of literals) in the body of rule r. Let $sub \in neg(r)$ be a negated subexpression in the body of a rule r. Rule r can be replaced with a transformed set of rules of the form $t(r, sub) = \{head(r) \leftarrow body(r) \setminus sub \cup \neg K \mid K \in sub)\}$. A rule set $\mathcal{R}$ can be reduced to $\mathcal{R} \setminus (\{r \mid r \in \mathcal{R}, neg(r) \neq \emptyset\} \cup (\bigcup_{\{r \mid r \in \mathcal{R}\}} \bigcup_{\{sub \mid sub \in neg(r)\}} t(r, sub))$.*

A.4 DECOMPOSE AND OUTER JOIN ON PRIMARY KEY

SMO: `DECOMPOSE TABLE` R `INTO` $S(A)$, $T(B)$ `ON PK`
Inverse: `OUTER JOIN TABLE` S, T `INTO` R `ON PK`

The decomposition using a foreign key is the simplest form of the `DECOMPOSE` SMO. The system-managed primary identifier p from the source table version remains unchanged in the two target table versions making the identification of corresponding tuples trivial. We construct the two target tables S and T with the following rules:

$$S(p, A) \leftarrow R(p, A, \_) \tag{A.23}$$
$$T(p, B) \leftarrow R(p, \_, B) \tag{A.24}$$

The inverse operation is an outer join. To fill the gaps potentially resulting from the outer join, we use the null value ω_R. Null values introduced from other SMOs or the user are assumed to be distinguishable from ω_R and treated as actual data. The outer join semantics are formalized in the following rule set for γ_{src}.

$$R(p, A, B) \leftarrow S(p, A), T(p, B) \tag{A.25}$$
$$R(p, A, \omega_R) \leftarrow S(p, A), \neg T(p, \_) \tag{A.26}$$
$$R(p, \omega_R, B) \leftarrow \neg S(p, \_), T(p, B) \tag{A.27}$$

Tuples in R having no values for either A or B need to be excluded from when reconstructing the decomposed table versions S and T. So, we end up with this rule set for the `DECOMPOSE` SMO on the primary key.

$\gamma_{\mathbf{trg}}$:

$$S(p, A) \leftarrow R(p, A, \_), A \neq \omega_R \tag{A.28}$$
$$T(p, B) \leftarrow R(p, \_, B), B \neq \omega_R \tag{A.29}$$

$\gamma_{\mathbf{src}}$:

$$R(p, A, B) \leftarrow S(p, A), T(p, B) \tag{A.30}$$
$$R(p, A, \omega_R) \leftarrow S(p, A), \neg T(p, \_) \tag{A.31}$$
$$R(p, \omega_R, B) \leftarrow \neg S(p, \_), T(p, B) \tag{A.32}$$

A.4.1 $R_D = \gamma_{src}(\gamma_{trg}(R_D))$

Initially, we insert the data R_D at the source side and show that we can store it at the target side and read it again unchanged at the source side.

$\gamma_{\mathbf{trg}}(\mathbf{R_D})$:

$$S(p, A) \leftarrow R_D(p, A, \_), A \neq \omega_R \tag{A.33}$$
$$T(p, B) \leftarrow R_D(p, \_, B), B \neq \omega_R \tag{A.34}$$

Since we consider existing null values in A and B as part of the data, we set $A \neq \omega_R$ and $B \neq \omega_R$ in R_D (p,A,B). We can remove the not-null conditions from both rules, as they are now implicitly assumed for R_D.

$$S(p, A) \leftarrow R_D(p, A, \_) \tag{A.35}$$
$$T(p, B) \leftarrow R_D(p, \_, B) \tag{A.36}$$

By simply applying the rules of γ_{src} to γ_{trg} (Lemma 1), we obtain:

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{R_D}))$:

$$R_D(p, A, B) \leftarrow R_D(p, A, \_), R_D(p, \_, B) \tag{A.37}$$
$$R_D(p, A, \omega_R) \leftarrow R_D(p, A, \_), \neg(R_D(p, \_, B')) \tag{A.38}$$
$$R_D(p, \omega_R, B) \leftarrow \neg(R_D(p, A', \_)), R_D(p, \_, B) \tag{A.39}$$

The key p uniquely determines the values A and B according to Lemma 5, hence we can reduce $R_D(p, A, \_)$ and $(R_D(p, \_, B)$ to $R_D(p, A, B)$ with the help of Lemma 7.

$$R(p, A, B) \leftarrow R_D(p, A, B) \tag{A.40}$$
$$R(p, A, \omega_R) \leftarrow R_D(p, A, B'), \neg(R_D(p, A, B')) \tag{A.41}$$
$$R(p, \omega_R, B) \leftarrow \neg(R_D(p, A', B)), R_D(p, A', B) \tag{A.42}$$

According to Lemma 4, we can remove Rules A.41 and A.42 to finally satisfy the data independence condition.

$$R(p, A, B) \leftarrow R_D(p, A, B) \quad \square \tag{A.43}$$

A.4.2 $\{S_D, T_D\} = \gamma_{trg}(\gamma_{src}(S_D, T_D))$

To show that the second data independence criteria holds as well for the decomposition on p, we now assume data in the target table versions S and T with source side materialization. Analogously to R_D in the previous step, we can now assume $A \neq \omega_R$ in $S_D(p, A)$ as well as $B \neq \omega_R$ in $T_D(p, B)$

$\gamma_{\mathbf{src}}(\mathbf{S_D}, \mathbf{T_D})$:

$$R(p, A, B) \leftarrow S_D(p, A), T_D(p, B) \tag{A.44}$$
$$R(p, A, \omega_R) \leftarrow S_D(p, A), \neg T_D(p, \_) \tag{A.45}$$
$$R(p, \omega_R, B) \leftarrow \neg S_D(p, \_), T_D(p, B) \tag{A.46}$$

We now apply Lemma 1 to calculate the combined rule set for one round trip of the target-side data. If applying a rule head with ω_R for the attributes C, we add the condition $C = \omega_R$ to make this missing value explicit.

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{S_D}, \mathbf{T_D}))$:

$$S(p, A) \leftarrow S_D(p, A), T_D(p, \_), A \neq \omega_R \tag{A.47}$$
$$S(p, A) \leftarrow S_D(p, A), \neg T_D(p, \_), A \neq \omega_R \tag{A.48}$$
$$S(p, A) \leftarrow \neg S_D(p, \_), T_D(p, B), A = \omega_R, A \neq \omega_R \tag{A.49}$$

$$T(p, B) \leftarrow S_D(p, \_), T_D(p, B), B \neq \omega_R \tag{A.50}$$
$$T(p, B) \leftarrow S_D(p, A), \neg T_D(p, \_), B = \omega_R, B \neq \omega_R \tag{A.51}$$
$$T(p, B) \leftarrow \neg S_D(p, \_), T_D(p, B), B \neq \omega_R \tag{A.52}$$

We remove those not-null conditions that are implicitly guaranteed by our initial assumptions. Further, we remove the Rules A.41 and A.42 with contradicting conditions according to Lemma 4. For instance $A = \omega_R \wedge A \neq \omega_R$ will never be true. So, we obtain the following rule set as intermediate result:

$$S(p, A) \leftarrow S_D(p, A), T_D(p, \_) \tag{A.53}$$
$$S(p, A) \leftarrow S_D(p, A), \neg T_D(p, \_) \tag{A.54}$$

$$T(p, B) \leftarrow S_D(p, \_), T_D(p, B) \tag{A.55}$$
$$T(p, B) \leftarrow \neg S_D(p, \_), T_D(p, B) \tag{A.56}$$

According to Lemma 3, we combine the two rules respectively and finally obtain:

$$S(p, A) \leftarrow S_D(p, A) \tag{A.57}$$
$$T(p, B) \leftarrow T_D(p, B) \quad \square \tag{A.58}$$

A.5 DECOMPOSE AND OUTER JOIN ON A FOREIGN KEY

SMO: `DECOMPOSE TABLE` R `INTO` $S(A)$, $T(B)$ `ON FK` t
Inverse: `OUTER JOIN TABLE` S, T `INTO` R `ON FK` t

When decomposing a table while creating a new foreign key, we have to generate new identifiers. Given we cut away the addresses from a persons table, we eliminate all duplicates in the new address table, assign a new identifier to each address, and finally add a foreign key column to the new persons table. We introduce functions that calculate new, globally unique identifiers, for given data. E.g. the function $id_T(B)$ returns a new identifier that uniquely identifies the payload data B in the table T. If the data B already exists, the function returns the previously assign identifier. If B did not exist in T before the id function returns a new globally unique identifier.

Intuitively, we want to decompose a source table version into two target table versions, each with a subset of the source table's columns. We eliminate arising duplicates in the second target table version T and create new identifiers p. These newly generated identifiers are then used in the first target table version S for the newly created foreign key column, as expressed with the two Datalog rules for γ_{trg}:

$$T(t, B) \leftarrow R(\_, \_, B), t = id_T(B) \tag{A.59}$$
$$S(p, A, t) \leftarrow R(p, A, B), T(t, B) \tag{A.60}$$

The original source table R is reconstructed by joining along the generated foreign key in γ_{src}. Obviously, the relationship between the original key p and the generated foreign key t is lost for source side materialization; therefore we introduce the auxiliary table ID_T that keeps exactly those two variables and is defined with the following rules in γ_{src}:

$$R(p, A, B) \leftarrow S(p, A, t), T(t, B) \tag{A.61}$$
$$ID_T(p, t) \leftarrow S(p, \_, t), T(t, \_) \tag{A.62}$$

Since the inverse of the decomposition is an outer join, we extend γ_{src} with the outer join semantics both for the data table R and the auxiliary table ID_T:

$$R(p, A, B) \leftarrow S(p, A, t), T(t, B) \tag{A.63}$$
$$R(p, A, \omega) \leftarrow S(p, A, \omega) \tag{A.64}$$
$$R(t, \omega, B) \leftarrow \neg S(\_, \_, t), T(t, B) \tag{A.65}$$
$$ID_T(p, t) \leftarrow S(p, \_, t), T(t, \_) \tag{A.66}$$
$$ID_T(p, \omega) \leftarrow S(p, \_, \omega) \tag{A.67}$$
$$ID_T(t, t) \leftarrow \neg S(\_, \_, t), T(t, \_) \tag{A.68}$$

The last two rules, for R and ID_T respectively, cover the cases that either no foreign key is given in S or a tuple in T is not matched by a foreign key. The rule bodies for R and ID_T are identical, however, the rule heads of ID_T merely store the key information. For reconstructing the target side data from the materialized source side, we have to extend γ_{trg} accordingly. Therefore, we end up with the following mapping semantics for the decomposition with a foreign key:

$\gamma_{\mathbf{trg}}$:

$$T(t, B) \leftarrow R(p, \_, B), ID_T(p, t) \tag{A.69}$$
$$T(t, B) \leftarrow R(p, \_, B), \neg ID_T(p, t), t = id_T(B) \tag{A.70}$$

$$S(p, A, t) \leftarrow R(p, A, \_), ID_R(p, t) \tag{A.71}$$
$$S(p, A, \omega) \leftarrow R(p, A, \_), ID_T(p, \omega) \tag{A.72}$$
$$S(p, A, t) \leftarrow R(p, A, B), \neg ID_T(p, \_), T(t, B) \tag{A.73}$$

$\gamma_{\mathbf{src}}$:

$$R(p, A, B) \leftarrow S(p, A, t), T(t, B) \tag{A.74}$$
$$R(p, A, \omega) \leftarrow S(p, A, \omega) \tag{A.75}$$
$$R(t, \omega, B) \leftarrow \neg S(\_, \_, t), T(t, B) \tag{A.76}$$
$$ID_T(p, t) \leftarrow S(p, \_, t), T(t, \_) \tag{A.77}$$
$$ID_T(p, \omega) \leftarrow S(p, \_, \omega) \tag{A.78}$$
$$ID_T(t, t) \leftarrow \neg S(\_, \_, t), T(t, \_) \tag{A.79}$$

A.5.1 $R_D = \gamma_{src}(\gamma_{trg}(R_D))$

According to Condition 6.1 for the data independence of materialized SMOs, any source-side data in R_D that is mapped to the decomposed target side and read back, survives this round trip completely and correctly. Let us start with the rules γ_{trg} that map the source data to the target side. Since ID_T is empty, we apply Lemma 2 to reduce the γ_{trg} rule set. Further, we use Lemma 1 to apply the Rule A.70 for T in the body of Rule A.73, which result in the following two rules:

$$T(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B) \tag{A.80}$$
$$S(p, A, t) \leftarrow R_D(p, A, B), R_D(\_, \_, B), t = id_T(B) \tag{A.81}$$

We can further reduce the latter rule based on Lemma 7.

$$T(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B) \tag{A.82}$$
$$S(p, A, t) \leftarrow R_D(p, A, B), t = id_T(B) \tag{A.83}$$

We will now apply γ_{src} to these reduced rules using Lemma 9.

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{R_D}))$:

$$R(p, A, B) \leftarrow R_D(p, A, B), t = id_T(B), R_D(\_, \_, B), t = id_T(B) \tag{A.84}$$
$$R(p, A, \omega) \leftarrow R_D(p, A, \omega), t = id_T(\omega) \tag{A.85}$$
$$R(t, \omega, B) \leftarrow \neg(R(\_, \_, B'), t = id_T(B')), R_D(\_, \_, B), t = id_T(B) \tag{A.86}$$
$$ID_T(p, t) \leftarrow R_D(p, A, B'), t = id_T(B'), R_D(\_, \_, B''), t = id_T(B'') \tag{A.87}$$
$$ID_T(p, \omega) \leftarrow R_D(p, \_, \omega), t = id_T(\omega) \tag{A.88}$$
$$ID_T(t, t) \leftarrow \neg(R_D(\_, \_, B'), t = id_T(B')), R_D(\_, \_, B''), t = id_T(B'') \tag{A.89}$$

Now, we will further simplify this rule set. There are rules with the two literals $t = id_T(B'), t = id_T(B'')$. So we can set $t = id_T(B') = id_T(B'')$. According to the definition of the $t = id_T(B)$ function, the resulting key t uniquely identifies the data B, hence $B' = B''$.

$$R(p, A, B) \leftarrow R_D(p, A, B), t = id_T(B), R_D(\_, \_, B), t = id_T(B) \tag{A.90}$$
$$R(p, A, \omega) \leftarrow R_D(p, A, \omega), t = id_T(\omega) \tag{A.91}$$
$$R(t, \omega, B) \leftarrow \neg(R_D(\_, \_, B), t = id_T(B)), R_D(\_, \_, B), t = id_T(B) \tag{A.92}$$
$$ID_T(p, t) \leftarrow R_D(p, A, B), t = id_T(B), R_D(\_, \_, B), t = id_T(B) \tag{A.93}$$
$$ID_T(p, \omega) \leftarrow R_D(p, \_, \omega), t = id_T(\omega) \tag{A.94}$$
$$ID_T(t, t) \leftarrow \neg(R_D(\_, \_, B), t = id_T(B)), R_D(\_, \_, B), t = id_T(B) \tag{A.95}$$

Further, we eliminate the calculation of the $id_T(B)$ in those rule, which do not use it in any other literal or in the head according to Lemma 6. Further we eliminate all redundant literals using Lemma 7.

$$R(p, A, B) \leftarrow R_D(p, A, B) \tag{A.96}$$
$$R(p, A, \omega) \leftarrow R_D(p, A, \omega) \tag{A.97}$$
$$R(t, \omega, B) \leftarrow \neg(R_D(\_, \_, B), t = id_T(B)), R(\_, \_, B), t = id_T(B) \tag{A.98}$$
$$ID_T(p, t) \leftarrow R_D(p, A, B), t = id_T(B) \tag{A.99}$$
$$ID_T(p, \omega) \leftarrow R_D(p, \_, \omega) \tag{A.100}$$
$$ID_T(t, t) \leftarrow \neg(R_D(\_, \_, B), t = id_T(B)), R_D(\_, \_, B), t = id_T(B) \tag{A.101}$$

To reduce Rule A.98, we use Lemma 10 to split it into two rules:

$$R(t, \omega, B) \leftarrow \neg R_D(\_, \_, B), R(\_, \_, B), t = id_T(B) \tag{A.102}$$
$$R(t, \omega, B) \leftarrow \neg(t = id_T(B)), R_D(\_, \_, B), t = id_T(B) \tag{A.103}$$

Obviously both rules cannot be satisfied, since there are contradicting literals and conditions (Lemma 4). Rule A.101 follows the same reductions as Rule A.98.

$$R(p, A, B) \leftarrow R_D(p, A, B) \tag{A.104}$$
$$R(p, A, \omega) \leftarrow R_D(p, A, \omega) \tag{A.105}$$
$$ID_T(p, t) \leftarrow R_D(p, A, B), t = id_T(B) \tag{A.106}$$
$$ID_T(p, \omega) \leftarrow R_D(p, \_, \omega) \tag{A.107}$$

Since ω is just a special value, which the attributes B may have, we can further reduce the rule set with the help of Lemma 8 and finally obtain the following rules:

$$R(p, A, B) \leftarrow R_D(p, A, B) \tag{A.108}$$
$$ID_T(p, t) \leftarrow R_D(p, A, B), t = id_T(B) \quad \square \tag{A.109}$$

Projecting the outcomes to the data tables, again satisfies our data independence Conditions 6.2 and 6.1. Hence, no matter whether the SMO is virtualized or materialized, both the source and the target side behave like common single-schema databases. Storing data in R implicitly generates new values to the auxiliary table ID_T, which is intuitive: we need to store the assigned identifiers for the target version to ensure repeatable reads on those generated identifiers.

A.5.2 $\{S_D, T_D\} = \gamma_{trg}(\gamma_{src}(S_D, T_D))$

For the opposite direction, we apply the γ_{trg} mapping to the γ_{src} mapping. For the input data in S_D and T_D, we enforce the foreign key constraint, hence $S_D(p, A, t), T_D(t, \_)$ needs to hold. Otherwise we assume $S(p, A, \omega)$. As there are no auxiliary tables in the rules' bodies of γ_{src}, there are no reductions possible at this stage.

$\gamma_{\mathbf{src}}(\mathbf{S_D}, \mathbf{T_D})$:

$$R(p, A, B) \leftarrow S_D(p, A, t), T_D(t, B) \tag{A.110}$$
$$R(p, A, \omega) \leftarrow S_D(p, A, \omega) \tag{A.111}$$
$$R(t, \omega, B) \leftarrow \neg S_D(\_, \_, t), T_D(t, B) \tag{A.112}$$
$$ID_T(p, t) \leftarrow S_D(p, \_, t), T_D(t, \_) \tag{A.113}$$
$$ID_T(p, \omega) \leftarrow S_D(p, \_, \omega) \tag{A.114}$$
$$ID_T(t, t) \leftarrow \neg S_D(\_, \_, t), T_D(t, \_) \tag{A.115}$$

Before we apply the rules γ_{trg}, we first reduce the γ_{trg} rule set. The rule bodies for R and ID_T in γ_{src} are equivalent, hence for each tuple in $R(p, \_)$ there is also one tuple $ID_T(p, \_)$. So, based on Lemma 4 we eliminate Rule A.70 and Rule A.73 and obtain a reduced rule set for γ_{trg}:

$$T(t, B) \leftarrow R(p, \_, B), ID_T(p, t) \tag{A.116}$$

$$S(p, A, t) \leftarrow R(p, A, \_), ID_T(p, t) \tag{A.117}$$
$$S(p, A, \omega) \leftarrow R(p, A, \_), ID_T(p, \omega) \tag{A.118}$$

When applying Lemma 1 in the next step, we apply only those rules for ID_T that can match the pattern according to Lemma 4. Particularly, we exclude the case that we set a value for $t \neq \omega$, but apply $ID_T(p, \omega)$. This, we obtain:

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{S_D}, \mathbf{T_D}))$:

$$T(t, B) \leftarrow S_D(p, \_, t), T(t, B), S_D(p, \_, t), T_D(t, \_) \tag{A.119}$$
$$T(t, B) \leftarrow S_D(p, \_, t), T_D(t, B), \neg S_D(\_, \_, t), T(t, \_) \tag{A.120}$$
$$T(t, B) \leftarrow \neg S_D(\_, \_, t), T_D(t, B), S_D(p, \_, t), T_D(t, \_) \tag{A.121}$$
$$T(t, B) \leftarrow \neg S_D(\_, \_, t), T_D(t, B), \neg S_D(\_, \_, t), T(t, \_) \tag{A.122}$$
$$S(p, A, t) \leftarrow S_D(p, A, t), T_D(t, \_), S_D(p, \_, t), T_D(t, \_) \tag{A.123}$$
$$S(p, A, t) \leftarrow S_D(p, A, t), T_D(t, \_), S_D(p, \_, t), t = \omega \tag{A.124}$$
$$S(p, A, t) \leftarrow S_D(p, A, \omega), S_D(p, \_, t), T_D(t, \_) \tag{A.125}$$
$$S(p, A, t) \leftarrow S_D(p, A, \omega), S_D(p, \_, t), t = \omega \tag{A.126}$$
$$S(p, A, \omega) \leftarrow S_D(p, A, t), T_D(t, \_), S_D(p, \_, t), t = \omega \tag{A.127}$$
$$S(p, A, \omega) \leftarrow S_D(p, A, \omega), S_D(p, \_, t), t = \omega \tag{A.128}$$

According to Lemma 4, we remove Rule A.120 and A.121 due to the contradicting literals on S_D. Further, we eliminate Rule A.124, A.125, and A.127 that require the primary key of a relation to be ω, since this can never be true. Additionally, we eliminate all subsumed literals according to Lemma 7.

$$T(t, B) \leftarrow S_D(p, \_, t), T_D(t, B) \tag{A.129}$$
$$T(t, B) \leftarrow \neg S_D(\_, \_, t), T_D(t, B) \tag{A.130}$$
$$S(p, A, t) \leftarrow S_D(p, A, t), T_D(t, \_) \tag{A.131}$$
$$S(p, A, t) \leftarrow S_D(p, A, \omega), t = \omega \tag{A.132}$$
$$S(p, A, \omega) \leftarrow S_D(p, A, t), T_D(t, \_) \tag{A.133}$$
$$S(p, A, \omega) \leftarrow S_D(p, A, \omega) \tag{A.134}$$

Using Lemma 3, we reduce the rules for T to already match our data independence criteria. Further, Lemma 8 allows to eliminate all duplicated and subsumed rules for S, so we obtain the following rule set:

$$T(t, B) \leftarrow T_D(t, B) \tag{A.135}$$
$$S(p, A, t) \leftarrow S_D(p, A, t), T_D(t, \_) \tag{A.136}$$
$$S(p, A, \omega) \leftarrow S_D(p, A, \omega) \tag{A.137}$$

Initially, we assumed that the foreign key constraint is enforces for the tuples in S. The defined foreign key constraint matches exactly the definition of S in Rules A.136 and A.136. Consequently, Lemma 7 allows us to further reduce the rule set for $\gamma_{trg}(\gamma_{src}(S_D, T_D))$.

$$T(t, B) \leftarrow T_D(t, B) \tag{A.138}$$
$$S(p, A, t) \leftarrow S_D(p, A, t) \quad \square \tag{A.139}$$

Now, both data independence conditions are satisfied for the decomposition and outer join along a foreign key.

A.6 DECOMPOSE AND OUTER JOIN ON CONDITION

SMO: `DECOMPOSE TABLE` R `INTO` $S(A)$, $T(B)$ `ON` $c(A,B)$
Inverse: `OUTER JOIN TABLE` S, T `INTO` R `ON` $c(A,B)$

To e.g. normalize a table that holds books and authors ($N : M$), we can either use two subsequent `DECOMPOSE ON FK` to maintain the relationship between books and authors, or—if the new evolved version just needs the list of authors and the list of books—we simply split them giving up the relationship. In the following, we provide rules for the latter case. Basically, we use the same rule set as for the inverse inner join on a condition (Section A.8) extended with the outer join and key-generation functionality introduced for the outer join along a foreign key (Section A.5). The formal evaluation of the data independence is just alike, so we do not elaborate on the details now.

$\gamma_{\mathbf{trg}}$:

$$S(s,A) \leftarrow R(r,A,\_), ID_{src}(r,s,\_) \tag{A.140}$$
$$S(s,A) \leftarrow R(r,A,\_), \neg ID_{src}(r,\_,\_), A \neq \omega_R, s = id_S(A) \tag{A.141}$$
$$S(r,A) \leftarrow R(r,A,\_), \neg ID_{src}(r,\_,\_), A = \omega_R \tag{A.142}$$
$$T(t,B) \leftarrow R(r,\_,B), ID_{src}(r,\_,t) \tag{A.143}$$
$$T(t,B) \leftarrow R(r,\_,B), \neg ID_{src}(r,\_,\_), B \neq \omega_R, t = id_T(B) \tag{A.144}$$
$$T(r,B) \leftarrow R(r,\_,B), \neg ID_{src}(r,\_,\_), B = \omega_R \tag{A.145}$$
$$ID_{trg}(r,s,t) \leftarrow R(r,A,B), S(s,A), T(t,B) \tag{A.146}$$
$$R^{-}(s,t) \leftarrow \neg R(\_,A,B), S(s,A), T(t,B), c(A,B) \tag{A.147}$$

$\gamma_{\mathbf{src}}$:

$$R_{tmp}(r,A,B) \leftarrow S(s,A), T(t,B), ID_{src}(r,s,t) \tag{A.148}$$
$$R_{tmp}(r,A,B) \leftarrow S(s,A), T(t,B), c(A,B), \neg R^{-}(s,t), \neg ID_{src}(\_,s,t), r = id_R(A,B) \tag{A.149}$$
$$ID_{src}(r,s,t) \leftarrow S(s,A), T(t,B), c(A,B), R_{tmp}(r,A,B) \tag{A.150}$$
$$ID_{src}(r,s,t) \leftarrow ID_{trg}(r,s,t) \tag{A.151}$$
$$R(r,A,B) \leftarrow R_{tmp}(r,A,B) \tag{A.152}$$
$$R(s,A,\omega_R) \leftarrow S(s,A), \neg ID_{trg}(\_,s,\_) \tag{A.153}$$
$$R(t,\omega_R,B) \leftarrow T(t,B), \neg ID_{trg}(\_,\_,t) \tag{A.154}$$

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{S_D}, \mathbf{T_D}))$:

$$R(r,A,B) \leftarrow R_D(r,A,B) \tag{A.155}$$
$$ID_{src}(r,s,t) \leftarrow R_D(r,A,B), s = id_S(A), t = id_T(B) \quad \square \tag{A.156}$$

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{R_D}))$:

$$S(s,A) \leftarrow S_D(s,A) \tag{A.157}$$
$$T(t,B) \leftarrow T_D(t,B) \tag{A.158}$$
$$ID_{trg}(r,s,t) \leftarrow S_D(s,A), T_D(t,B), c(A,B), r = id_R(A,B) \quad \square \tag{A.159}$$

The Conditions 6.2 and 6.1 for data independence are satisfied. For repeatable reads, the auxiliary table ID stores the generated identifiers independently of the chosen materialization.

A.7 INNER JOIN ON PRIMARY KEY

SMO: `JOIN TABLE` R, S `INTO` T `ON PK`

An inner join on the primary key follows a 1:1 relationship on the tuple level. Hence, we will not replicate any data nor will we remove any duplicates. The naïve mapping function for γ_{trg} is:

$$R(p, A, B) \leftarrow S(p, A), T(p, B) \tag{A.160}$$

The straight forward mapping γ_{src} for reconstruction of the source table is:

$$S(p, A) \leftarrow R(p, A, \_) \tag{A.161}$$
$$T(p, B) \leftarrow R(p, \_, B) \tag{A.162}$$

Obviously, all tuples in S or R that do not find a join partner are lost after one round trip. To prevent from this information loss, we merely need two auxiliary tables to store those tuples from the input table versions that do not match with a join partner. The auxiliary tables S^+ and T^+ store the whole lost tuples for target side materialization that can be added after one round trip in the γ_{src} mapping. In sum, the inner join on a primary key is described with the following rule sets:

$\gamma_{\mathbf{trg}}$:

$$R(p, A, B) \leftarrow S(p, A), T(p, B) \tag{A.163}$$
$$S^+(p, A) \leftarrow S(p, A), \neg T(p, \_) \tag{A.164}$$
$$T^+(p, B) \leftarrow \neg S(p, \_), T(p, B) \tag{A.165}$$

$\gamma_{\mathbf{src}}$:

$$S(p, A) \leftarrow R(p, A, \_) \tag{A.166}$$
$$S(p, A) \leftarrow S^+(p, A) \tag{A.167}$$
$$T(p, B) \leftarrow R(p, \_, B) \tag{A.168}$$
$$T(p, B) \leftarrow T^+(p, B) \tag{A.169}$$

A.7.1 $\{S_D, T_D\} = \gamma_{src}(\gamma_{trg}(S_D, T_D))$

At first, we show that data at the source side can be written and read correctly with target side materialization. Since γ_{trg} does not contain any auxiliary tables in the rule bodies, we cannot reduce γ_{trg} at this point, so the following rules define the target side materialization.

$\gamma_{\mathbf{trg}}(\mathbf{S_D}, \mathbf{T_D})$:

$$R(p, A, B) \leftarrow S_D(p, A), T_D(p, B) \tag{A.170}$$
$$S^+(p, A) \leftarrow S_D(p, A), \neg T_D(p, \_) \tag{A.171}$$
$$T^+(p, B) \leftarrow \neg S_D(p, \_), T_D(p, B) \tag{A.172}$$

By applying γ_{src} to γ_{trg} according to Lemma 1, we obtain the following rule set:

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{S_D}, \mathbf{T_D}))$:

$$S(p, A) \leftarrow S_D(p, A), T_D(p, \_) \tag{A.173}$$
$$S(p, A) \leftarrow S_D(p, A), \neg T_D(p, \_) \tag{A.174}$$
$$T(p, B) \leftarrow S_D(p, \_), T_D(p, B) \tag{A.175}$$
$$T(p, B) \leftarrow \neg S_D(p, \_), T_D(p, B) \tag{A.176}$$

Using Lemma 3 we can combine both the rules for S and for T, so we finally end up with the satisfied data independence condition:

$$S(p, A) \leftarrow S_D(p, A) \tag{A.177}$$
$$T(p, B) \leftarrow T_D(p, B) \quad \square \tag{A.178}$$

A.7.2 $R_D = \gamma_{trg}(\gamma_{src}(R_D))$

In the opposite direction—source side materialization of data in R—we can reduce γ_{src} since the auxiliary tables S^+ and T^+ are initially empty (Lemma 2).

$\gamma_{\mathbf{src}}(\mathbf{R_D})$:

$$S(p, A) \leftarrow R_D(p, A, \_) \tag{A.179}$$
$$T(p, B) \leftarrow R_D(p, \_, B) \tag{A.180}$$

Applying γ_{trg} to these rules results in the following rule set according to Lemma 1:

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{R_D}))$:

$$R(p, A, B) \leftarrow R_D(p, A, \_), R_D(p, \_, B) \tag{A.181}$$
$$S^+(p, A) \leftarrow R_D(p, A, \_), \neg R_D(p, \_, \_) \tag{A.182}$$
$$T^+(p, B) \leftarrow \neg R_D(p, \_, \_), R_D(p, \_, B) \tag{A.183}$$

Using Lemma 5 we can reduce the rule body of Rule A.181 to $R_D(p, A, B)$. The key p uniquely identifies the data for the attributes A and B. Further, the bodies of the Rules A.182 and A.183 will never be satisfied, since there are contradicting literals (Lemma 4).

$$R(p, A, B) \leftarrow R_D(p, A, B) \quad \square \tag{A.184}$$

Since both data independence conditions hold for the inner join on a primary key, we have formally shown its data independence.

A.8 INNER JOIN ON CONDITION

SMO: `JOIN TABLE` R, S `INTO` T `ON` $c(A, B)$

When performing an inner join on an arbitrary condition we potentially follow a $n : m$ relationship. For the newly created tuples, we have to generate new identifiers p as well. The initial definition of the γ_{trg} mapping merely combines the tuples from S and T according to the specified condition. Additionally, we need to calculate a new identifier r and immediately store it in the auxiliary table ID_{trg} along with the keys of the two joint source tables to ensure repeatable reads.

$$R(r, A, B) \leftarrow S(s, A), T(t, B), c(A, B), r = id_R(A, B) \tag{A.185}$$

$$ID_{trg}(r, s, t) \leftarrow S(s, A), T(t, B), c(A, B), R(r, A, B) \tag{A.186}$$

For reconstructing the disjoint tables from the joint target table version, we need to project R to the respective columns and add the previously stored primary key information in γ_{src}.

$$S(s, A) \leftarrow R(r, A, \_), ID_{trg}(r, s, \_) \tag{A.187}$$

$$T(t, B) \leftarrow R(r, \_, B), ID_{trg}(r, \_, t) \tag{A.188}$$

Further, adding tuples at the disjoint source side is not the only occasion when we have to create new identifiers. Also when writing at the target side to the joint table R, we need to generate new identifiers for the disjoint source table versions and immediately store the relationship to the original identifier in the auxiliary table ID_{src}, hence γ_{src} is extended by the following three rules:

$$S(s, A) \leftarrow R(r, A, \_), \neg ID_{trg}(r, s, \_), s = id_S(A) \tag{A.189}$$

$$T(t, B) \leftarrow R(r, \_, B), \neg ID_{trg}(r, \_, t), t = id_T(B) \tag{A.190}$$

$$ID_{src}(r, s, t) \leftarrow R(r, A, B), S(s, A), T(t, B) \tag{A.191}$$

Obviously, we only generate new identifiers at the target side, if there are no existing ones for the respective payload data. Hence, we add two rules to γ_{trg} to use already existing identifiers.

$$R(r, A, B) \leftarrow S(s, A), T(t, B), ID_{src}(r, s, t) \tag{A.192}$$

$$ID_{trg}(r, s, t) \leftarrow ID_{src}(r, s, t) \tag{A.193}$$

Since we now consider the inner join, for target side materialization we also need auxiliary tables that store those tuples that did not find a join partner. We use the auxiliary table S^+ and T^+ that are specified in γ_{trg} as follows:

$$S^+(s, A) \leftarrow S(s, A), \neg ID_{trg}(\_, s, \_) \tag{A.194}$$

$$T^+(t, B) \leftarrow T(t, B), \neg ID_{trg}(\_, \_, t) \tag{A.195}$$

For reconstruction of those outer join tuples, we add two further rules to γ_{src} that add the non-matched tuples from the auxiliary tables:

$$S(s, A) \leftarrow S^+(s, A) \tag{A.196}$$

$$T(t, B) \leftarrow T^+(t, B) \tag{A.197}$$

Finally, the opposite can happen as well—given source side materialization of the disjoint table versions, there can be tuples in the disjoint tables that match the join condition but have e.g. been explicitly deleted at the target side. To prevent this information loss, we use the auxiliary table R^- that stores exactly those tuples and make sure that they are excluded when reading data at the target side:

$$R^-(s, t) \leftarrow \neg R(\_, A, B), S(s, A), T(t, B), c(A, B) \tag{A.198}$$

Summing up our discussion so far, we finally use the following rule set to describe the inner join on an arbitrary condition:

$\gamma_{\mathbf{trg}}$:

$$R(r, A, B) \leftarrow S(s, A), T(t, B), ID_{src}(r, s, t) \tag{A.199}$$
$$R(r, A, B) \leftarrow S(s, A), T(t, B), c(A, B), \neg R^{-}(s, t), \neg ID_{src}(\_, s, t), r = id_R(A, B) \tag{A.200}$$
$$ID_{trg}(r, s, t) \leftarrow S(s, A), T(t, B), c(A, B), R(r, A, B) \tag{A.201}$$
$$ID_{trg}(r, s, t) \leftarrow ID_{src}(r, s, t) \tag{A.202}$$
$$S^{+}(s, A) \leftarrow S(s, A), \neg ID_{trg}(\_, s, \_) \tag{A.203}$$
$$T^{+}(t, B) \leftarrow T(t, B), \neg ID_{trg}(\_, \_, t) \tag{A.204}$$

$\gamma_{\mathbf{src}}$:

$$S(s, A) \leftarrow R(r, A, \_), ID_{trg}(r, s, \_) \tag{A.205}$$
$$S(s, A) \leftarrow R(r, A, \_), \neg ID_{trg}(r, s, \_), s = id_S(A) \tag{A.206}$$
$$S(s, A) \leftarrow S^{+}(s, A) \tag{A.207}$$

$$T(t, B) \leftarrow R(r, \_, B), ID_{trg}(r, \_, t) \tag{A.208}$$
$$T(t, B) \leftarrow R(r, \_, B), \neg ID_{trg}(r, \_, t), t = id_T(B) \tag{A.209}$$
$$T(t, B) \leftarrow T^{+}(t, B) \tag{A.210}$$

$$ID_{src}(r, s, t) \leftarrow R(r, A, B), S(s, A), T(t, B) \tag{A.211}$$
$$R^{-}(s, t) \leftarrow \neg R(\_, A, B), S(s, A), T(t, B), c(A, B) \tag{A.212}$$

A.8.1 $\{R_D, S_D\} = \gamma_{src}(\gamma_{trg}(R_D, S_D))$

Now, we formally evaluate the first data independence criteria, checking whether source side data R_D and S_D can be stored at the target side and retrieved completely and correctly. Since the auxiliary tables R^{-} and ID_{src} are empty, we use Lemma 2 to reduce the γ_{trg} rule set. This leaves only one rule for R and ID_{trg} respectively. Further, we remove the empty literals R^{-} and ID_{src} in Rule A.200.

$\gamma_{\mathbf{trg}}(\mathbf{S_D}, \mathbf{T_D})$:

$$R(r, A, B) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.213}$$
$$ID_{trg}(r, s, t) \leftarrow S_D(s, A), T_D(t, B), c(A, B), R(r, A, B) \tag{A.214}$$
$$S^{+}(s, A) \leftarrow S_D(s, A), \neg ID_{trg}(\_, s, \_) \tag{A.215}$$
$$T^{+}(t, B) \leftarrow T_D(t, B), \neg ID_{trg}(\_, \_, t) \tag{A.216}$$

Now, we apply Rule A.214 to Rule A.213 based on Lemma 1 and immediately eliminate duplicate literals according to Lemma 7.

$$R(r, A, B) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.217}$$
$$ID_{trg}(r, s, t) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.218}$$
$$S^{+}(s, A) \leftarrow S_D(s, A), \neg ID_{trg}(\_, s, \_) \tag{A.219}$$
$$T^{+}(t, B) \leftarrow T_D(t, B), \neg ID_{trg}(\_, \_, t) \tag{A.220}$$

Following Lemma 9, we further apply the rules for S^+ and T^+ to ID_{trg}.

$$R(r, A, B) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.221}$$
$$ID_{trg}(r, s, t) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.222}$$
$$S^+(s, A) \leftarrow S_D(s, A), \neg(S_D(s, A'), T_D(t', B'), c(A', B'), r' = id_R(A', B')) \tag{A.223}$$
$$T^+(t, B) \leftarrow T_D(t, B), \neg(S_D(s', A'), T_D(t, B'), c(A', B'), r' = id_R(A', B')) \tag{A.224}$$

In the rule set γ_{src}, the rules for T follow the same idea as the rules for S. For brevity, we focus on the rules for S in detail and apply the same reductions for T as well. Using Lemma 1, we combine and reduce the rule sets of γ_{trg} and γ_{src}.

$\gamma_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{S_D}, \mathbf{T_D}))$:

$$\begin{aligned} S(s, A) \leftarrow & S_D(s, A), T_D(t, B'), c(A, B'), r = id_R(A, B'), \\ & S_D(s, A''), T_D(t'', B''), c(A'', B''), r = id_R(A'', B'') \end{aligned} \tag{A.225}$$
$$\begin{aligned} S(s, A) \leftarrow & S_D(s, A), T_D(t, B'), c(A, B'), r = id_R(A, B'), \\ & \neg(S_D(s, A''), T_D(t'', B''), c(A'', B''), r = id_R(A'', B'')), s = id_S(A) \end{aligned} \tag{A.226}$$
$$S(s, A) \leftarrow S_D(s, A), \neg(S_D(s, A'), T_D(t', B'), c(A', B'), r' = id_R(A', B')) \tag{A.227}$$
$$ID_{src}(r, s, t) \leftarrow S_D(s', A), T_D(t', B), c(A, B), r = id_R(A, B), S(s, A), T(t, B) \tag{A.228}$$
$$\begin{aligned} R^-(s, t) \leftarrow & \neg(S_D(s', A), T_D(t', B), c(A, B), r' = id_R(A, B)), \\ & S(s, A), T(t, B), c(A, B) \end{aligned} \tag{A.229}$$

To reduce the rules for S, we first reduce the rules individually, before considering their union. First, we focus on **Rule A.225** for S: Using Lemma 5 we set $A = A''$, because the key s uniquely identifies the data in S_D. Further, we set $B' = B''$, since $r = id_R(A, B') = id_R(A'', B'')$ due to the definition of the id function. Further, the identifiers t and t'' are computed, but never referenced again. Hence, we eliminated them according to Lemma 6.

$$\begin{aligned} S(s, A) \leftarrow & S_D(s, A), T_D(\_, B), c(A, B), r = id_R(A, B), \\ & S_D(s, A), T_D(\_, B), c(A, B), r = id_R(A, B) \end{aligned} \tag{A.230}$$

Now, we summarize equal literals according to Lemma 7 and remove $r = id_R(A, B)$ since it is not referenced again (Lemma 6). So we finally reduced Rule A.225 to

$$S(s, A) \leftarrow S_D(s, A), T_D(\_, B), c(A, B) \tag{A.231}$$

Second, we focus on the second **Rule A.226** for S:

$$\begin{aligned} S(s, A) \leftarrow & S_D(s, A), T_D(t, B'), c(A, B'), r = id_R(A, B'), \\ & \neg(S_D(s, A''), T_D(t'', B''), c(A'', B''), r = id_R(A'', B'')), s = id_S(A) \end{aligned} \tag{A.232}$$

Again, we set $A = A''$ and $B' = B''$ since they are determined by equal identifiers (Lemma 5).

$$\begin{aligned} S(s, A) \leftarrow & S_D(s, A), T_D(\_, B), c(A, B), r = id_R(A, B), \\ & \neg(S_D(s, A), T_D(\_, B), c(A, B), r = id_R(A, B)), s = id_S(A) \end{aligned} \tag{A.233}$$

By applying Lemma 10, we obtain four rules—due to Lemma 4 none of them is satisfiable and we can safely remove Rule A.226 from the rule set. Together with third **Rule A.227** we now obtain the reduced rule set for S:

$$S(s, A) \leftarrow S_D(s, A), T_D(\_, B), c(A, B) \tag{A.234}$$
$$S(s, A) \leftarrow S_D(s, A), \neg(S_D(s, A'), T_D(t', B'), c(A', B'), r' = id_R(A', B')) \tag{A.235}$$

The identifiers r' and t' are never referenced, hence we eliminate them based on Lemma 6. Further $A = A'$, since the identifier s uniquely determines A and A' according to Lemma 5.

$$S(s, A) \leftarrow S_D(s, A), T_D(\_, B), c(A, B) \tag{A.236}$$
$$S(s, A) \leftarrow S_D(s, A), \neg(S_D(s, A), T_D(\_, B'), c(A, B')) \tag{A.237}$$

We further apply Lemma 10 to Rule A.237 to obtain:

$$S(s, A) \leftarrow S_D(s, A), T_D(\_, B), c(A, B) \tag{A.238}$$
$$S(s, A) \leftarrow S_D(s, A), \neg S_D(s, A) \tag{A.239}$$
$$S(s, A) \leftarrow S_D(s, A), \neg(T_D(\_, B'), c(A, B')) \tag{A.240}$$

Rule A.239 is never satisfied and can be removed according to Lemma 4. We combine Rule A.238 and Rule A.240 using Lemma 3. So, we finally reduced the three Rules A.225 to A.227.

$$S(s, A) \leftarrow S_D(s, A) \tag{A.241}$$

Accordingly, we obtain the same for T.

$$T(t, B) \leftarrow T_D(t, B) \tag{A.242}$$

Further, we apply the remaining Rules A.228 and A.229 to these reduced rules for S and T with the help of Lemma 9.

$$ID_{src}(r, s, t) \leftarrow S_D(s', A), T_D(t', B), c(A, B), r = id_R(A, B), S_D(s, A), T_D(t, B) \tag{A.243}$$
$$R^-(s, t) \leftarrow \neg(S_D(s', A), T_D(t', B), c(A, B), r' = id_R(A, B)), S_D(s, A), T_D(t, B), c(A, B) \tag{A.244}$$

We replace those attributes, which are only referenced once with _ or remove respective calculations according to Lemma 6.

$$ID_{src}(r, s, t) \leftarrow S_D(\_, A), T_D(\_, B), c(A, B), r = id_R(A, B), S_D(s, A), T_D(t, B) \tag{A.245}$$
$$R^-(s, t) \leftarrow \neg(S_D(\_, A), T_D(\_, B), c(A, B)), S_D(s, A), T_D(t, B), c(A, B) \tag{A.246}$$

Now, we remove subsumed or duplicate literals based on Lemma 7.

$$ID_{src}(r, s, t) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \tag{A.247}$$

We cannot further reduce this rule producing the identifiers auxiliary table. Obviously we always need to store the generated identifiers for all tuples that match the inner join. This is reasonable, since we have to make sure to always assign the same identifiers to the same data. Regarding the rules for R^-, we use Lemma 10 to obtain the three rules:

$$R^-(s, t) \leftarrow \neg S_D(\_, A), S_D(s, A), T_D(t, B), c(A, B) \tag{A.248}$$
$$R^-(s, t) \leftarrow \neg T_D(\_, B), S_D(s, A), T_D(t, B), c(A, B) \tag{A.249}$$
$$R^-(s, t) \leftarrow \neg c(A, B), S_D(s, A), T_D(t, B), c(A, B) \tag{A.250}$$

All three rules are not satisfiable according to Lemma 4. Hence we finally obtain

$$S(s, A) \leftarrow S_D(s, A) \tag{A.251}$$
$$T(t, B) \leftarrow T_D(t, B) \tag{A.252}$$
$$ID_{src}(r, s, t) \leftarrow S_D(s, A), T_D(t, B), c(A, B), r = id_R(A, B) \quad \square \tag{A.253}$$

A.8.2 $R_D = \gamma_{trg}(\gamma_{src}(R_D))$

To show that data independence also holds for the backward direction of the inner join, we now assume data given at the target side, which we store at source side. Initially, the auxiliary tables ID_{trg}, S^+, and T^+ are empty, hence we use Lemma 2 to reduce the γ_{src} rule set. Further, the identifier r referenced only once in the rule body and not in the head, hence we replace it with _.

$\gamma_{\mathbf{src}}(\mathbf{R_D})$:

$$S(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A) \tag{A.254}$$
$$T(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B) \tag{A.255}$$
$$ID_{src}(r, s, t) \leftarrow R_D(r, A, B), S(s, A), T(t, B) \tag{A.256}$$
$$R^-(s, t) \leftarrow \neg R_D(\_, A, B), S(s, A), T(t, B), c(A, B) \tag{A.257}$$

We apply the rules for ID_{src} and R^- to these reduced rules for S and T using lemma 1.

$$S(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A) \tag{A.258}$$
$$T(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B) \tag{A.259}$$
$$ID_{src}(r, s, t) \leftarrow R_D(r, A, B), R_D(\_, A, \_), s = id_S(A), R_D(\_, \_, B), t = id_T(B) \tag{A.260}$$
$$R^-(s, t) \leftarrow \neg R_D(\_, A, B), R_D(\_, A, \_), s = id_S(A), R_D(\_, \_, B), t = id_T(B), c(A, B) \tag{A.261}$$

We further remove subsumed literals, like e.g. $R(\_, A, \_)$ which is subsumed by $R(r, A, B)$, according to Lemma 7.

$$S(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A) \tag{A.262}$$
$$T(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B) \tag{A.263}$$
$$ID_{src}(r, s, t) \leftarrow R_D(r, A, B), s = id_S(A), t = id_T(B) \tag{A.264}$$
$$R^-(s, t) \leftarrow \neg R_D(\_, A, B), s = id_S(A), t = id_T(B), c(A, B) \tag{A.265}$$

We now apply the rules of γ_{trg}. Again, we make the evaluation easier to follow by using Lemma 9.

$\gamma_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{R_D}))$:

$$R(r,A,B) \leftarrow R_D(\_,A,\_), s = id_S(A), R_D(\_,\_,B), t = id_T(B), R_D(r,A',B'), s = id_S(A'), t = id_T(B') \tag{A.266}$$

$$R(r,A,B) \leftarrow R_D(\_,A,\_), s = id_S(A), R_D(\_,\_,B), t = id_T(B), c(A,B), \neg(\neg R_D(\_,A'',B''), s = id_S(A''), t = id_T(B''), c(A'',B'')), \neg(R_D(\_,A',B'), s = id_S(A'), t = id_T(B')), r = id_R(A,B) \tag{A.267}$$

$$ID_{trg}(r,s,t) \leftarrow R_D(\_,A,\_), s = id_S(A), R_D(\_,\_,B), t = id_T(B), c(A,B), R(r,A,B) \tag{A.268}$$

$$ID_{trg}(r,s,t) \leftarrow R_D(r,A',B'), s = id_S(A'), t = id_T(B') \tag{A.269}$$

$$S^+(s,A) \leftarrow R_D(\_,A,\_), s = id_S(A), \neg ID_{trg}(\_,s,\_) \tag{A.270}$$

$$T^+(t,B) \leftarrow R_D(\_,\_,B), t = id_T(B), \neg ID_{trg}(\_,\_,t) \tag{A.271}$$

To reduce Rule A.266, we set $A = A'$ and $B = B'$, since identifiers functions only return the same result for the same input values (Lemma 5). The same applies to Rule A.267 where $A = A' = A''$ and $B = B' = B''$. Further, we eliminate the calculations, since the results are not referenced again according to Lemma 6.

$$R(r,A,B) \leftarrow R_D(\_,A,\_), R_D(\_,\_,B), R_D(r,A,B) \tag{A.272}$$

$$R(r,A,B) \leftarrow R_D(\_,A,\_), R_D(\_,\_,B), c(A,B), \neg(\neg R_D(\_,A,B), c(A,B)), \neg R_D(\_,A,B), r = id_R(A,B) \tag{A.273}$$

Using Lemma 10 we split the second rule into two rules.

$$R(r,A,B) \leftarrow R_D(\_,A,\_), R_D(\_,\_,B), R_D(r,A,B) \tag{A.274}$$

$$R(r,A,B) \leftarrow R_D(\_,A,\_), R_D(\_,\_,B), c(A,B), R_D(\_,A,B), \neg R_D(\_,A,B), r = id_R(A,B) \tag{A.275}$$

$$R(r,A,B) \leftarrow R_D(\_,A,\_), R_D(\_,\_,B), c(A,B), \neg c(A,B), \neg R_D(\_,A,B), r = id_R(A,B) \tag{A.276}$$

We remove two literals from Rule A.274, since they are subsumed by $R_D(r,A,B)$ w.r.t. Lemma 7. Further, Rules A.275 and A.276 can be eliminated since they contain contradicting literals according to Lemma 4.

$$R(r,A,B) \leftarrow R_D(r,A,B) \tag{A.277}$$

After reducing the rules for R we apply the result to the two remaining rules for ID_{trg}.

$$ID_{trg}(r,s,t) \leftarrow R_D(\_,A,\_), s = id_S(A), R_D(\_,\_,B), t = id_T(B), c(A,B), R_D(r,A,B) \tag{A.278}$$

$$ID_{trg}(r,s,t) \leftarrow R_D(r,A',B'), s = id_S(A'), t = id_T(B') \tag{A.279}$$

The first rule can be reduced by eliminating those literals that are subsumed by $R_D(r,A,B)$ with the help of Lemma 7.

$$ID_{trg}(r,s,t) \leftarrow R_D(r,A,B), s = id_S(A), t = id_T(B), c(A,B) \tag{A.280}$$

$$ID_{trg}(r,s,t) \leftarrow R_D(r,A',B'), s = id_S(A'), t = id_T(B') \tag{A.281}$$

Since the latter rule is subsumed by the first rule according to Lemma 8, we can remove it as well.

$$ID_{trg}(r, s, t) \leftarrow R_D(r, A, B), s = id_S(A), t = id_T(B) \tag{A.282}$$

So, the auxiliary table ID_{trg} always holds the generated identifiers for any tuple. Finally, we reduce the two remaining rules for S^+ and T^+:

$$S^+(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A), \neg ID_{trg}(\_, s, \_) \tag{A.283}$$
$$T^+(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B), \neg ID_{trg}(\_, \_, t) \tag{A.284}$$

We apply them to the reduced ID_{trg} rules with Lemma 9.

$$S^+(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A), \neg(R_D(r', A', B'), s = id_S(A'), t' = id_T(B')) \tag{A.285}$$
$$T^+(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B), \neg(R_D(r', A', B'), s' = id_S(A'), t = id_T(B')) \tag{A.286}$$

According to Lemma 6, unreferenced calculations and attributes may be eliminated.

$$S^+(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A), \neg(R_D(\_, A', \_), s = id_S(A')) \tag{A.287}$$
$$T^+(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B), \neg(R_D(\_, \_, B'), t = id_T(B')) \tag{A.288}$$

Using Lemma 10, we eliminate the negation and obtain two rules respectively.

$$S^+(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A), \neg R_D(\_, A', \_) \tag{A.289}$$
$$S^+(s, A) \leftarrow R_D(\_, A, \_), s = id_S(A), \neg s = id_S(A') \tag{A.290}$$
$$T^+(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B), \neg R_D(\_, \_, B') \tag{A.291}$$
$$T^+(t, B) \leftarrow R_D(\_, \_, B), t = id_T(B), \neg t = id_T(B') \tag{A.292}$$

Lemma 4 states that all four rules bodies can never be satisfied, since each one contains contradicting literals and conditions. Thus, we have reduced γ_{trg} (γ_{src}) to finally show that the data independence is always guaranteed.

$$R(r, A, B) \leftarrow R_D(r, A, B) \tag{A.293}$$
$$ID_{trg}(r, s, t) \leftarrow R_D(r, A, B), s = id_S(A), t = id_T(B) \quad \square \tag{A.294}$$

B

ASYMMETRIES FOR DATA ACCESS PROPAGATION THROUGH SMOS

As discussed in Section 7.3.1, the performance overhead for the different SMOs of BiDEL is asymmetric. This means that the overhead for data access propagation in one direction is higher than in the opposite direction. For the adviser, it is important to know these characteristics—we therefore analyze them in detail. In Section 7.3.1, we focus the discussion on the `ADD COLUMN` SMO and analyze the empiric measurements in detail. The following charts show the same measurements for the remaining SMOs. The findings from these additional measurements are already summarized in Section 7.3.1. As there are many different extents of the `JOIN` and the `DECOMPOSE` SMO, we exemplarily show the inner join on a condition and the decomposition to two tables that are linked by a foreign key, here.

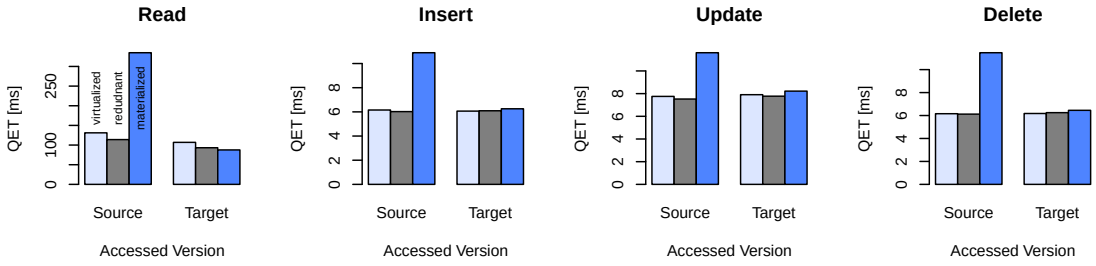


Figure B.1: Accessing table versions of `DROP COLUMN` SMO with different materializations.

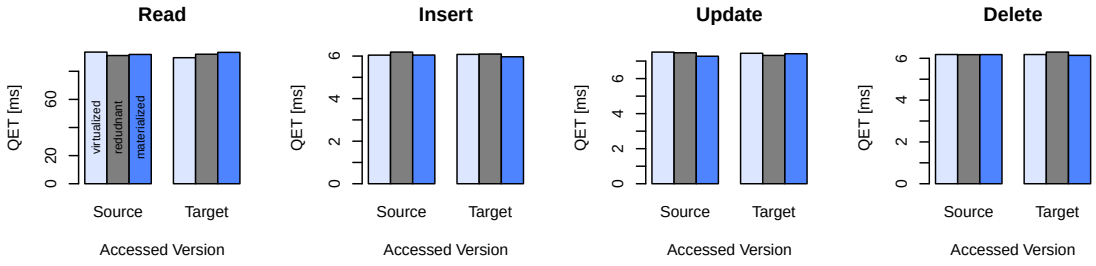


Figure B.2: Accessing table versions of `RENAME COLUMN` SMO with different materializations.

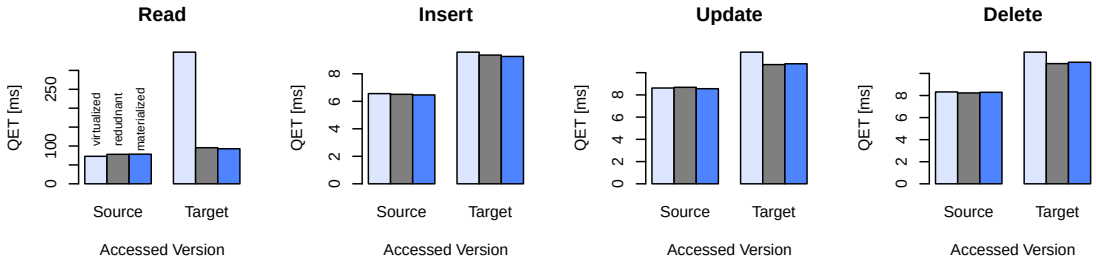


Figure B.3: Accessing table versions of inner `JOIN` SMO on a condition with different materializations.

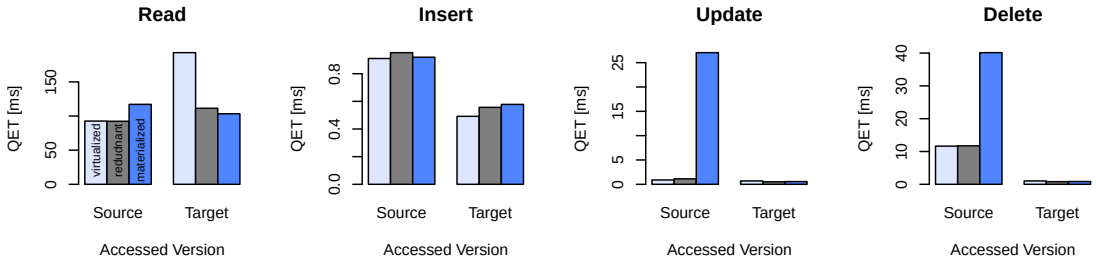


Figure B.4: Accessing table versions of `DECOMPOSE` SMO on a FK with different materializations.

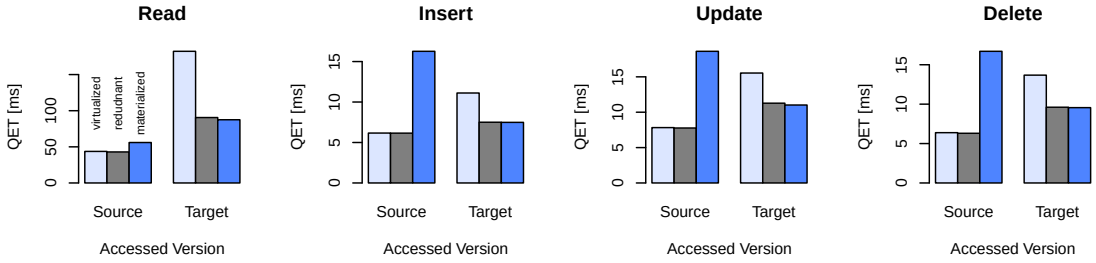


Figure B.5: Accessing table versions of `MERGE` SMO with different materializations.

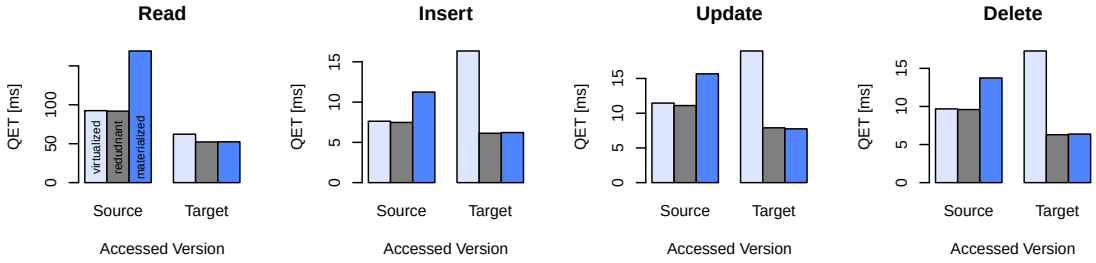


Figure B.6: Accessing table versions of `PARTITION` SMO with different materializations.

C

SCALING BEHAVIOR OF ACCESS PROPAGATION THROUGH SMOS

In Section 7.3.2, we measure and estimate the performance for propagating data accesses through up to two SMOs. However, the shown charts are restricted to the combinations of any SMO with the `ADD COLUMN` SMO. The following figures provide the remaining measurements. We do see the same effects here. There is a significant difference between accessing data locally or propagating it through an SMO, so there is significant optimization potential the DBA can use by adapting the physical table schema to the current workload. Further, the DBA can do so, without fearing a penalty for propagating data accesses through sequences of SMOs. The performance for propagating data through two SMOs is just the intuitively expected linear combination—hence the performance behaves predictably.

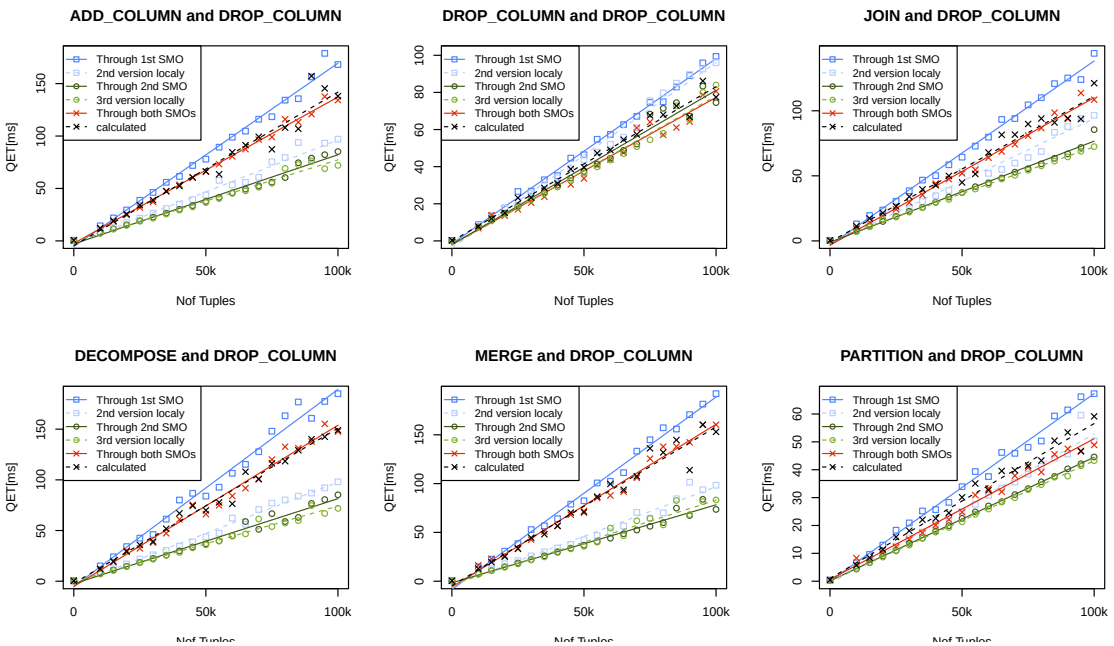


Figure C.1: Performance for all SMOs combined with the `DROP COLUMN` SMO.

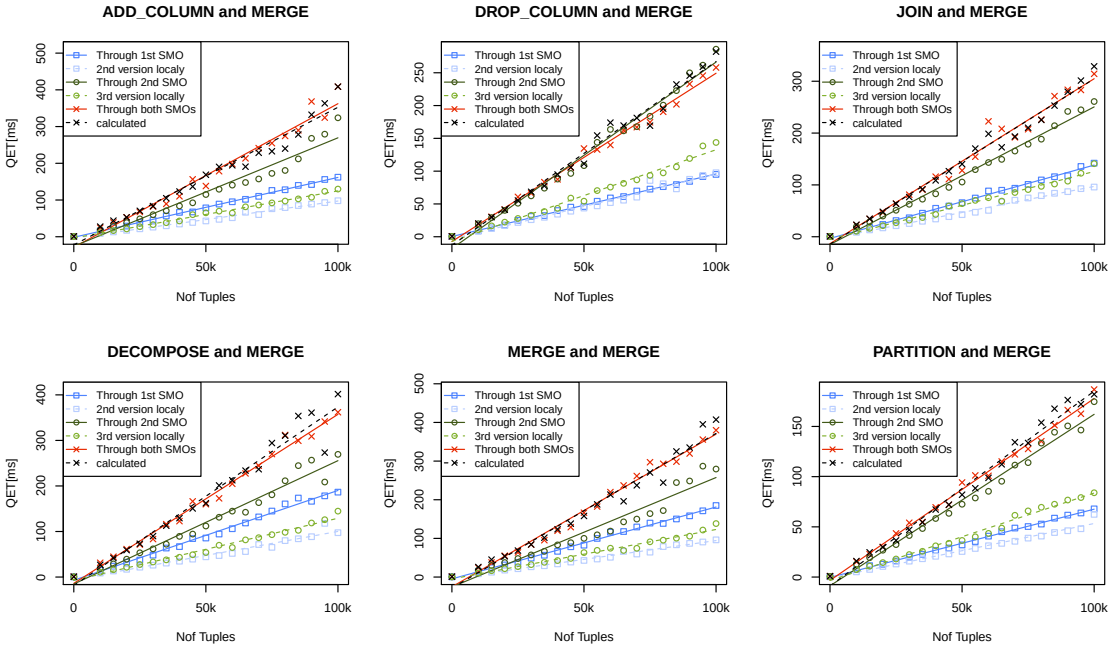


Figure C.2: Performance for all SMOs combined with the MERGE SMO.

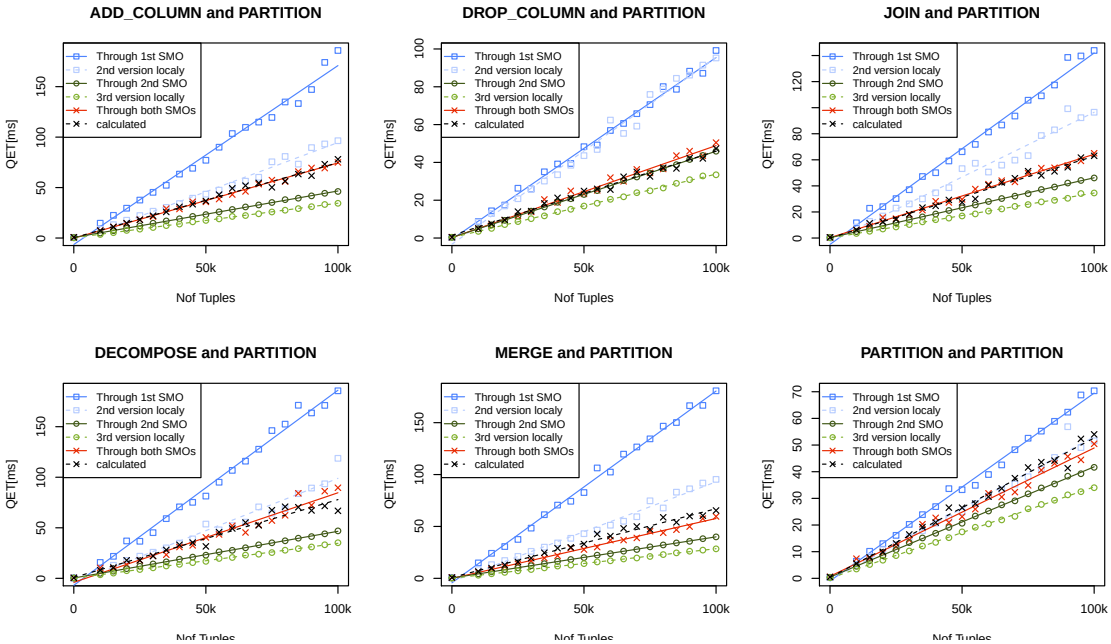


Figure C.3: Performance for all SMOs combined with the PARTITION SMO.

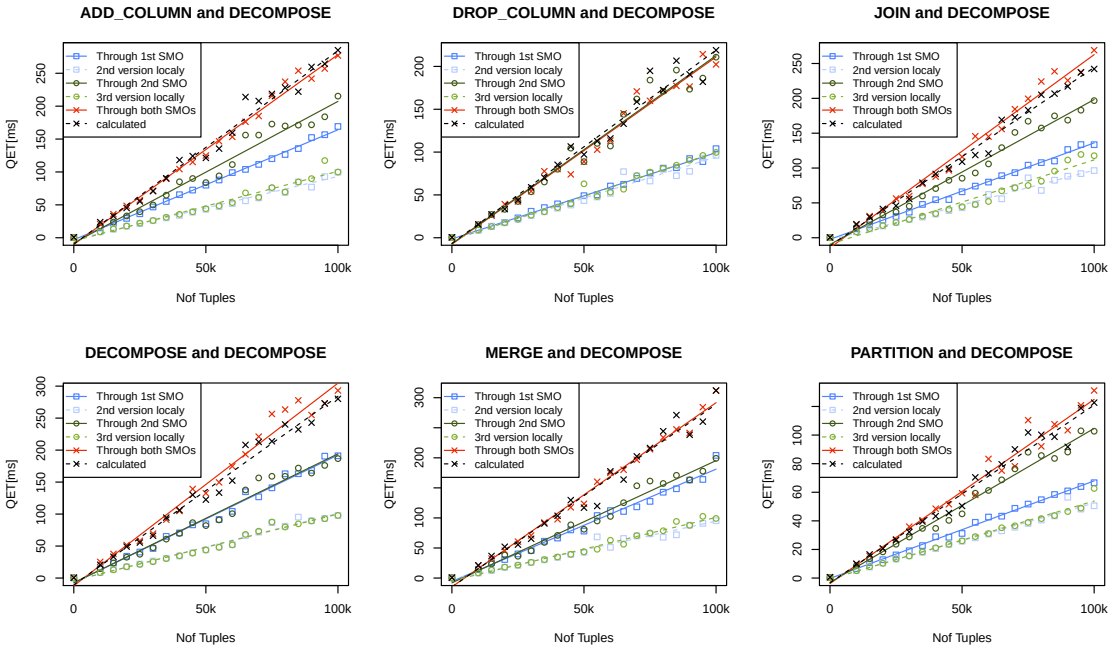


Figure C.4: Performance for all SMOs combined with the DECOMPOSE SMO.

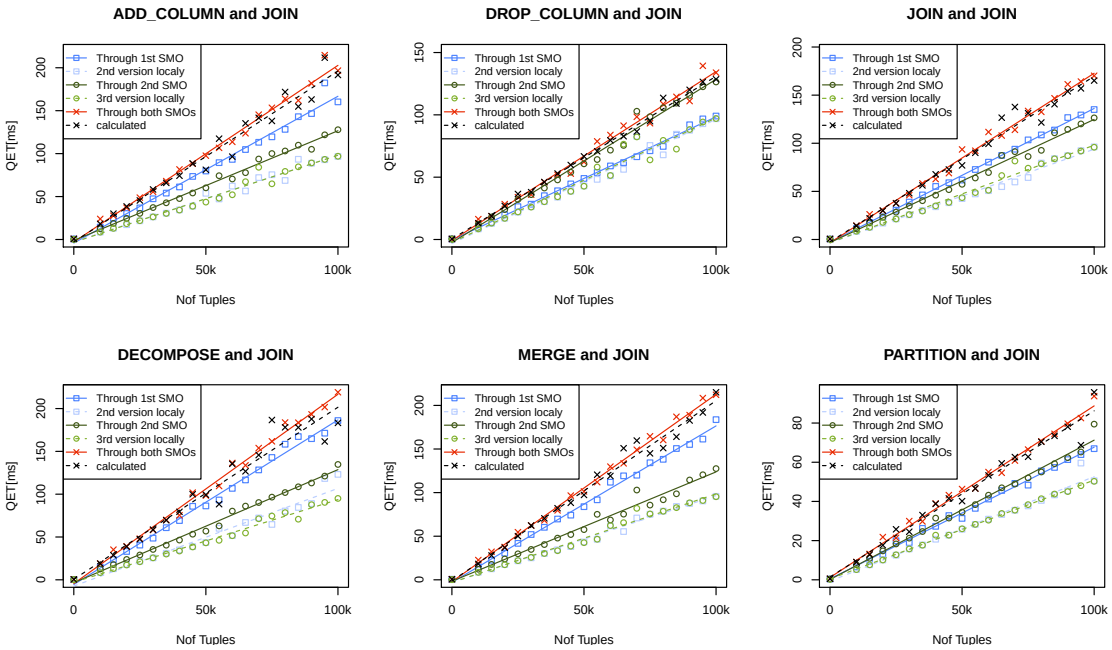


Figure C.5: Performance for all SMOs combined with the JOIN SMO.

D

EVALUATION OF INVERDA'S COST MODEL

In Section 7.3.4, we create a cost model to provide the adviser's optimizer with fast estimates of the costs for executing a given workload on a given system with a given materialization. We evaluate the estimation error of the developed cost model in Section 7.6.1. We conduct the empiric evaluation for all combinations of two subsequent SMO—however, the shown figures are restricted to those pairs that have an `ADD COLUMN`, a `JOIN`, or a `MERGE` as second SMO, so the following figures show the same measurements for the remaining combinations as well. The results are already discussed in Section 7.6.1.

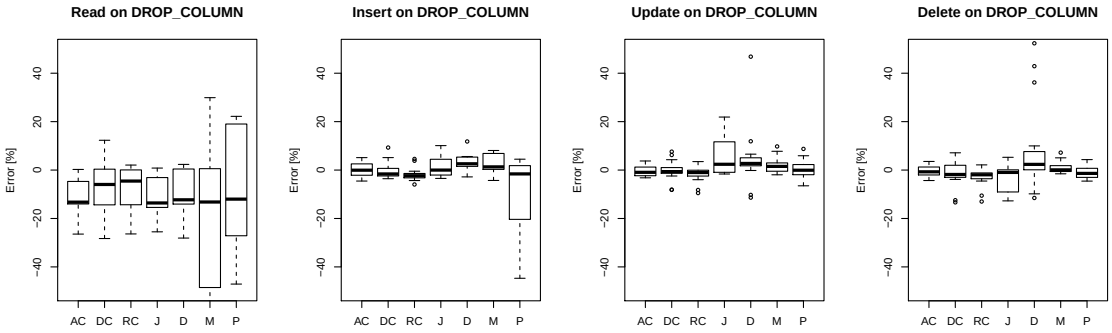


Figure D.1: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a `DROP COLUMN` SMO.

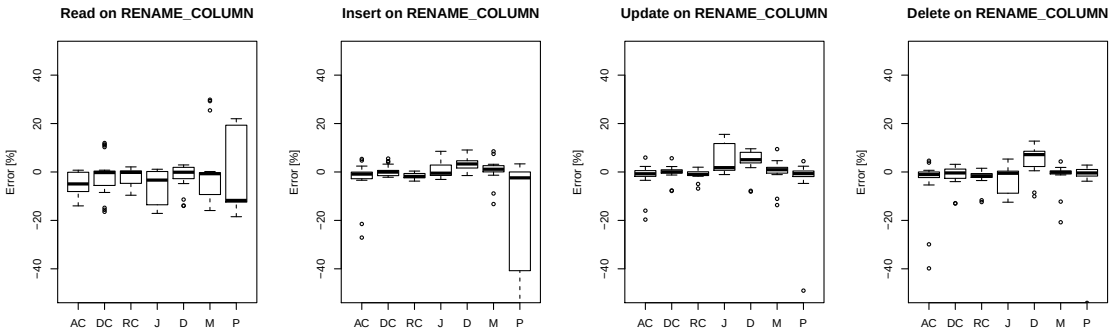


Figure D.2: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a `RENAME COLUMN` SMO.

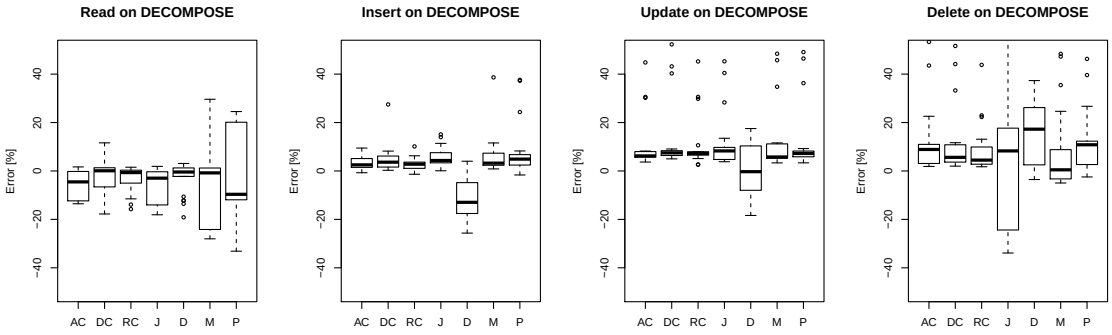


Figure D.3: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a DECOMPOSE SMO.

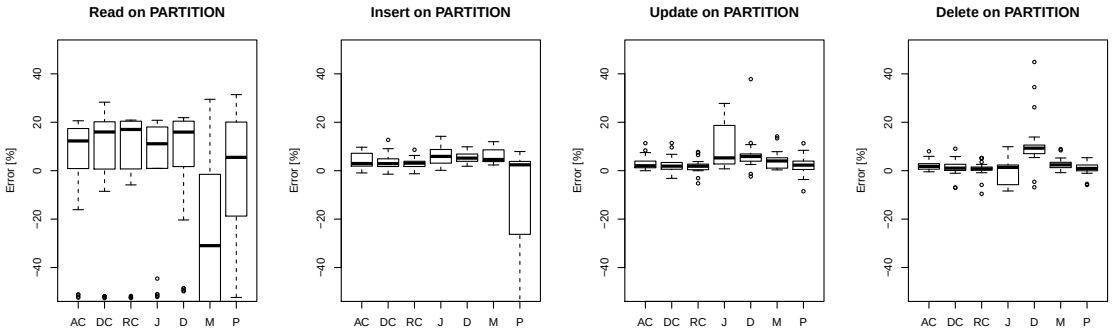


Figure D.4: Error of the cost estimation for read and write operations in an evolved database with two subsequent SMOs, with the second being a PARTITION SMO.

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, September 28, 2017

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-074-6

AALBORG UNIVERSITY PRESS