Multi-schema-version data management

data independence in the twenty-first century

Herrmann, Kai; Voigt, Hannes; Pedersen, Torben Bach; Lehner, Wolfgang

Published in:
V L D B Journal

DOI (link to publication from Publisher):
10.1007/s00778-018-0508-7

Creative Commons License
Unspecified

Publication date:
2018

Document Version
Accepted author manuscript, peer reviewed version

Multi-schema-version data management

data independence in the twenty-first century

Herrmann, Kai; Voigt, Hannes; Pedersen, Torben Bach; Lehner, Wolfgang

Published in:
V L D B Journal

DOI (link to publication from Publisher):
10.1007/s00778-018-0508-7

Creative Commons License
Unspecified

Publication date:
2018

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Herrmann, K., Voigt, H., Pedersen, T. B., & Lehner, W. (2018). Multi-schema-version data management: data independence in the twenty-first century. V L D B Journal, 27(4), 547-571. https://doi.org/10.1007/s00778-018-0508-7

Noname manuscript No.
(will be inserted by the editor)

Multi-Schema-Version Data Management

Data Independence in the 21st Century

Kai Herrmann · Hannes Voigt · Torben Bach Pedersen · Wolfgang Lehner

Received: date / Accepted: date

Abstract Agile software development allows us to continuously evolve and run a software system. However, this is not possible in databases, as established methods are very expensive, error-prone, and far from agile. We present InVerDa, a Multi-Schema-Version Database Management System (MSVDB) for agile database development. MSVDBs realize co-existing schema versions within one database, where each schema version behaves like a regular single-schema database and write operations are propagated between schema versions. Developers use a relationally complete and bidirectional database evolution language (BiDEL) to easily evolve existing schema versions to new ones. BiDEL scripts are more robust, orders of magnitude shorter, and cause only a small performance overhead compared to handwritten SQL scripts. We formally guarantee data independence: no matter how the data of the co-existing schema versions is physically materialized, each schema version is guaranteed to behave like a regular database. Since, the chosen physical materialization significantly determines the overall performance, we equip database administrators with an advisor that proposes an optimized materialization for the current workload, which can improve the performance by orders of magnitude compared to naïve solutions. To our best knowledge, we are the first to facilitate agile evolution of production databases with full support of co-existing schema versions and formally guaranteed data independence.

Keywords Database · Evolution · Data independence

Kai Herrmann · Hannes Voigt · Wolfgang Lehner
Technische Universität Dresden, Germany
E-mail: <firstname>.<lastname>@tu-dresden.de

Torben Bach Pedersen
Aalborg Universitet, Denmark
E-mail: tbp@cs.aau.dk

1 Introduction

Modern database management systems (DBMSes) lack proper support for co-existing schema versions within the same database. With today's realities in information systems—specifically agile development methods, code refactoring, stepwise deployment, short release cycles, varying update adoption time, legacy system support, etc.—such support becomes increasingly desirable. Development tools such as GIT, SVN, and Maven make it feasible to maintain multiple versions of an application and deploy and run several of these versions concurrently. The same is hard to do for a database though [6]. In enterprise information systems, databases feed hundreds of subsystems, connecting decades-old legacy systems with innovative analytics pipelines or brand new web front ends. All of these subsystems are typically run by different stakeholders with different development cycles and different upgrade constraints. Applying changes to a database schema will stretch out over time, hence the database schema versions of these subsystems have to be kept fully functional on the database side. Co-existing schema versions in a database are a reality and developers have to cope with it.

Unfortunately, current DBMSes do not support easy evolution with co-existing schema versions and essentially force developers to migrate a database completely at one go to a new schema version. Keeping old schema versions up and running for all database clients regardless of their adoption time is costly. Before and after a 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, e.g., view and trigger definitions within the database, propagation code in the application's database access layer, or ETL jobs for update propagation in a database replicated in different schema versions. We can automatically select views to be materialized to speed up a given workload [1] and we can maintain them efficiently [12]. However, this does not allow to migrate a database completely to a new version. In fact, co-existing schema versions require considerable development resources as data migration accounted for 31 % of IT project budgets in 2011 [17].

We introduce **Multi-Schema-Version Database Management Systems (MSVDBs)** to support agile database evolution, particularly the creation, management, and deployment of different schema versions to keep pace with agile software development. All schema versions co-exist in the MSVDB and applications can read and write data in any version concurrently; writes in one version are reflected in all other versions with best effort but each schema version acts like a regular single-schema database. The envisioned MSVDBs have full data independence: the database administrator (DBA) can easily configure the physical materialization of the co-existing schema versions to gain significant speedups without affecting the availability of any schema version. We specify the envisioned MSVDBs (Section 1.1), highlight our contributions (Section 1.2), and illustrate them with a user scenario (Section 1.3).

1.1 Multi-Schema-Version Database Systems

MSVDBs facilitate multi-schema-version data management. They manage multiple co-existing schema versions within the same database to support continuous database evolution as shown in Figure 1. New schema versions are evolved from existing ones using a simple **Database Evolution Language (DEL)**. DELs provide **Schema Modification Operations (SMOs)** that evolve both the schema and the data in intuitive and consistent steps, such as adding a column or partitioning a table. SMOs in an MSVDB are **bidirectional** to propagate data both forwards and backwards between schema versions. Each schema version consists of a set of *table versions*, which behave like regular tables but are part of the evolution history. Table versions in one schema version are evolved by SMOs to new table versions in the new schema version and MSVDBs store this evolution explicitly in 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 schema versions.

MSVDBs expose the data through multiple schema versions, while each schema version can be accessed by applications like a regular single-schema database.

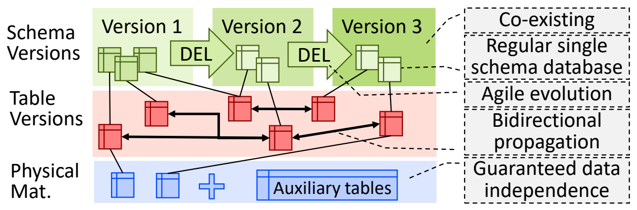


Fig. 1: Envisioned MSVDB.

MSVDBs ensure transaction and consistency guarantees for each single schema version. After developers create a new schema version, it becomes immediately available and **co-exists** with all other schema versions. Data written in any schema version is correctly visible when reading the same schema version again. To propagate data between schema versions both forwards and backwards, the DEL's SMOs need 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 with the best overall performance. This requires the freedom to easily adjust the physical materialization to a changing workload without affecting the logical schema versions, a.k.a **data independence**. Multiple logical schema versions represent the same conceptual data set (logical data independence) and the physical materialization of this data can be freely changed (physical data independence) without limiting accessibility of any schema version. Since most SMOs are not information-preserving, MSVDBs also identify and persist the otherwise lost information; this **auxiliary information** is persisted in *auxiliary tables*. MSVDBs guarantee that the auxiliary tables cover the whole conceptual data set and no data is lost. We can also retire old schema versions and remove them from the MSVDB—then, no additional data is managed for those versions any longer.

MSVDBs need only a subset of the table versions to be materialized. Generally, read operations are fastest when the accessed table versions are materialized. The further away the next materialized table version, the higher the overhead. Naïvely materializing every single table version implies a high overhead for writing and may hit memory limits, hence the envisioned MSVDBs also equip the DBA with an **advisor** that proposes an **optimized materialization** for the current workload.

In a word, the envisioned MSVDBs provide multiple co-existing schema versions upon one shared conceptual data set with an independent materialization. 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.

1.2 Contributions

In this article, we present the concepts of InVerDa (Integrated Versioning of Databases), a system that realizes the envisioned MSVDB. InVerDa lets developers create multiple schema versions within the same database using our DEL called BiDEL (Bidirectional DEL). BiDEL's distinguishing features are its relational completeness and its bidirectionality, which facilitate fast and easy creation of co-existing schema versions and robust data access propagation between them. We formally guarantee data independence, so developers can safely create and use new schema versions while the DBA can freely change the physical materialization—nobody needs to fear any data loss and can focus on the actual tasks. In detail, our contributions are:

Architectural blueprint We show how to integrate InVerDa into an existing DBMS to reuse established database functionality. To our best knowledge, we are the first to implement end-to-end support for MSVDBs. (Section 2)

Bidirectional and relationally complete DEL BiDEL is shown to be relationally complete and it enables bidirectional propagation of data accesses both forwards and backwards. Further, BiDEL requires orders of magnitude less code than evolutions and migrations manually written in SQL. (Section 3)

Co-existing schema versions We automatically generate 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. Delta code generation is fast ($<1\,\text{s}$) and its performance is comparable to handwritten delta code. (Section 4 and 6)

Data independence We formally guarantee that each schema version behaves like a regular single-schema database independently of the physical materialization. With a single-line command, DBAs can trigger the physical data movement and adapt all delta code, which improves the runtime performance and productivity by orders of magnitude. (Section 5)

Advisor for materialization The space of possible materializations can grow exponentially with the number of table versions and is hard to explore for a DBA. To this end, InVerDa comes with an advisor for the best materialization for the current workload under given space constraints. (Section 7)

The formal contributions are already published in conference papers [14, 15] and in a detailed PhD thesis [13]. The contribution of this article is to apply the formal results in an MSVDB to make them exploitable for both researchers and practitioners. We present the big picture and complete it with the new advisor.

1.3 User Story – The TasKy Example

We use the running example of a task management software called `TasKy` (Figure 2) to illustrate InVerDa. `TasKy` is a desktop application backed by one 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 with 1 being the most urgent. In the first release `TasKy`, we store all tasks in a single table `Task(author,task,prio)`. The physical schema matches the logical schema without any delta code and users begin to feed the database with tasks.

Creating new schema versions: After some weeks, we incorporate a third-party phone app called `Do!` for mobile access to the most urgent tasks. `Do!` uses a different database schema than `TasKy`: table `Todo(author,task)` contains only tasks of priority 1. Obviously, the initial schema version needs to stay alive for `TasKy`, which is broadly installed. Traditionally, we would implement a view to create an external schema fitting `Do!` and triggers to propagate new tasks from `Do!` to `TasKy`. This is highly error-prone and expensive, since we would have to manage the auxiliary information manually to not lose any data. InVerDa greatly simplifies the job as developers merely have to write this BiDEL script:

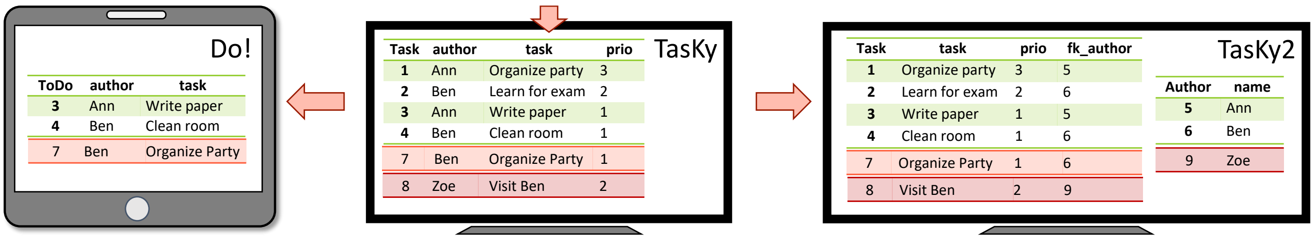
```
1: CREATE VERSION Do! FROM TasKy WITH
2:   PARTITION TABLE Task INTO Todo
       WITH 'prio=1';
3:   DROP COLUMN prio FROM Todo DEFAULT '1';
```

InVerDa automatically generates all delta code to provide the new schema version `Do!`. `Do!` contains a horizontal partition of `Task` with `prio=1` and the priority column being dropped. When a user inserts a new entry into `Todo`, this will automatically insert a corresponding task with priority 1 to `Task` in `TasKy`. Updates and deletions are propagated back to the `TasKy` schema in the same way. The `TasKy` data is immediately available to be read and written through the new `Do!` app by simply executing the BiDEL script. At this point, InVerDa has already simplified the developers' jobs significantly.

Rolling upgrade: We continuously refine `TasKy` and for the next release `TasKy2` we normalize the table `Task` to separate the authors in `Author`. For a stepwise roll-out of `TasKy2`, the old schema of `TasKy` has to remain accessible until all clients are updated. Again, InVerDa does the job when the developers run this BiDEL script:

```
1: CREATE 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



ToDo	author	task
3	Ann	Write paper
4	Ben	Clean room
7	Ben	Organize Party

Task	author	task	prio
1	Ann	Organize party	3
2	Ben	Learn for exam	2
3	Ann	Write paper	1
4	Ben	Clean room	1
7	Ben	Organize Party	1
8	Zoe	Visit Ben	2

Task	task	prio	fk_author
1	Organize party	3	5
2	Learn for exam	2	6
3	Write paper	1	5
4	Clean room	1	6
7	Organize Party	1	6
8	Visit Ben	2	9

Author	name
5	Ann
6	Ben
9	Zoe

Fig. 2: The exemplary evolution of TasKy.

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. Now, write operations to any of the three schema versions are propagated to all other schema versions. Figure 2 shows two examples: First, Ben uses the mobile Do! app to note that he will organize a party, which 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 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. Common SQL would require to implement this propagation and the management of the auxiliary information manually. BiDEL's SMOs carry enough information to generate all the required delta code automatically without any further interaction of the developer.

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 with a single line of code:

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

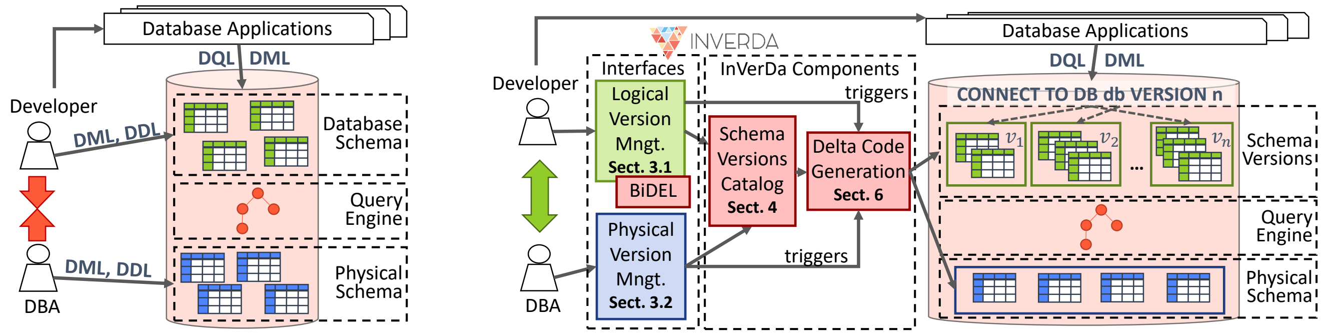
As a result, the schema version TasKy cannot be accessed or evolved any longer but both Do! and TasKy2 are still accessible like any other single-schema database.

Physical data migration: We have not migrated any data so far, hence it is still stored according to the initial TasKy schema. The delta code for accessing Do! and TasKy causes a significant performance overhead. The TasKy version has already been dropped and most users use TasKy2—the Do! version is still accessed but merely by a minority of users. Therefore, 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 them from messing up our data. With InVerDa, this takes merely a single line of code:

```
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. All schema versions stay available; read and write operations are merely propagated to a different set of physical tables, now. InVerDa can also materialize any subset of all table versions, which opens up a huge search space and significant optimization potential, but is hard to understand and utilize for DBAs. So, InVerDa includes an advisor that proposes an optimized set of physically materialized table versions for the current workload mix. With the click of a button, the DBA migrates the database accordingly.

Summary & Outline: InVerDa allows applications to continuously access all schema versions, it allows the developer to continuously develop the applications, and it allows the DBA to independently adapt the physical materialization to the current workload or even let InVerDa do this job. In Section 2, we present the general architecture of MSVDBs, such as InVerDa. In Section 3, we detail the user interfaces that allow us to create and manage schema versions in the schema versions catalog. The latter will be presented in Section 4. Having all managed schema versions in the catalog, we discuss their physical design in Section 5 with special focus on the formal guarantee of full data independence. This ensures that the DBA can freely move the data along the schema version history while all schema versions stay fully alive and behave like regular single-schema databases. Subsequently, we present the automatic delta code generation for the co-existing schema version on the chosen physical design in Section 6. In Section 7, we present a workload-dependent advisor for the physical materialization. We discuss related work in Section 9 and conclude the article in Section 10.



(a) Current architecture (b) Envisioned architecture

Fig. 3: Architecture of current and envisioned DBMS.

2 Architecture of MSVDBs

Starting with an analysis of current DBMSes, we will now propose an architectural blueprint for MSVDBs. In traditional DBMSes, as shown in Figure 3(a), developers use SQL-DDL statements to define the database's schema. When applications access data in the created schema, the schema catalog maps the 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. Developers use SQL-DDL to evolve the database schema and SQL-DML to evolve the currently existing data accordingly. Analogously, DBAs use SQL-DDL and SQL-DML to change the physical schema including data migration. 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 DBA is lost between the lines and whenever one of them evolves the database, the work of the respective other one most likely gets corrupted or invalidated. In summary, creating and managing co-existing schema versions is an error-prone and expensive challenge, since current DBMSes do not provide a good separation of developer and DBA concerns when it comes to fast evolution and multiple schema versions.

MSVDBs, 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 typically restricted to one schema version, which is directly mapped to physical tables in the storage.

Therefore, MSVDBs add support for fast evolution and for management of co-existing schema versions on a common relational DBMS. Figure 3(b) shows the architecture of our MSVDB InVerDa. InVerDa follows a three layered architecture with the logical schema versions at the upper layer, the physically materialized tables at the lower layer and a catalog in between to connect those layers. InVerDa's functionality is exposed via two interfaces: (1) the *logical version management* allows developers to create and manage multiple schema versions and (2) the *physical version management* allows the DBA to physically manage the materialization for all co-existing schema versions within the database. Further, InVerDa adds two components to facilitate multi-schema-version data management: (1) 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. (2) The *delta code generation* creates the schema versions based on the current physical materialization and also migrates the data. Delta code generation is either triggered by developers creating a new schema version or by the DBA changing the physical materialization.

In our prototypical implementation of InVerDa[1], the generated delta code for the co-existing schema versions are 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.

[1] Online demo available at www.inverda.de

Syntax:	`CREATE TABLE` R `(`c_1`,...,`c_n`)`	$\emptyset \leftrightarrow$ ▦
Semantics:	$\texttt{Add}(R, \pi_{c_1,\dots,c_n}(\emptyset));$	
Syntax:	`DROP TABLE` R	▦ $\leftrightarrow \emptyset$
Semantics:	$\texttt{Del}(R);$	
Syntax:	`RENAME TABLE` R `INTO` R'	▦ $\leftrightarrow$ ▦
Semantics:	$\texttt{Add}\big(R', R\big); \texttt{Del}(R);$	
Syntax:	`RENAME COLUMN` c `IN` R_i `TO` c'	▦ $\leftrightarrow$ ▦
Semantics:	$\texttt{Add}\big(R_{i+1}, \rho_{c'/c}(R_i)\big); \texttt{Del}(R_i);$	
Syntax:	`ADD COLUMN` c `AS` f`(`c_1`,...,`c_n`)` `INTO` R_i	▦ $\leftrightarrow$ ▦
Semantics:	$\texttt{Add}\big(R_{i+1}, \pi_{R_i.C \cup \{c \leftarrow f(c_1,\dots,c_n)\}}(R_i)\big); \texttt{Del}(R_i);$	
Syntax:	`DROP COLUMN` c `FROM` R_i `DEFAULT` f`(`c_1`,...,`c_n`)`	▦ $\leftrightarrow$ ▦
Semantics:	$\texttt{Add}\big(R_{i+1}, \pi_{R_i.C \setminus \{c\}}(R_i)\big); \texttt{Del}(R_i);$	
Syntax:	`DECOMPOSE TABLE` R `INTO` S `(`s_1`,...,`s_n`)` `[,` T `(`t_1`,...,`t_m`)` `ON (PK` $\mid$ `FK` fk $\mid$ $cond$`)]`	▦ $\leftrightarrow$ ▦▦
Semantics:	$\texttt{Add}(S, \pi_{s_1,\dots,s_n}(R));$ $[\texttt{Add}(T, \pi_{t_1,\dots,t_m}(R))];$ $\texttt{Del}(R);$	
Syntax:	`[OUTER] JOIN TABLE` R`,` S `INTO` T `ON (PK` $\mid$ `FK` fk $\mid$ $cond$`)`	▦▦ $\leftrightarrow$ ▦
Semantics:	if [OUTER] then $\texttt{Add}(T, R ⟗_{cond} S)$ else $\texttt{Add}(T, R \bowtie_{cond} S); \texttt{Del}(R); \texttt{Del}(S);$	
Syntax:	`PARTITION TABLE` R `INTO` S `WITH` $cond_S$ `[,` T `WITH` $cond_T$`]`	▦ $\leftrightarrow$ ▦▦
Semantics:	$\texttt{Add}(S, \sigma_{cond_S}(R));$ $[\texttt{Add}(T, \sigma_{cond_T}(R))];$ $\texttt{Del}(R);$	
Syntax:	`MERGE TABLE` R `(`$cond_R$`)`, S `(`$cond_S$`)` `INTO` T	▦▦ $\leftrightarrow$ ▦
Semantics:	$\texttt{Add}\big(T, \pi_{R.C \cup \{\omega \rightarrow a_i \mid a_i \in S.C \setminus R.C\}}(R) \cup \pi_{S.C \cup \{\omega \rightarrow a_i \mid a_i \in R.C \setminus S.C\}}(S)\big); \texttt{Del}(R); \texttt{Del}(S);$	

Table 1: Syntax and semantics of BiDEL operations.

Summary: We add interfaces for logical and physical management of schema versions that are represented in a schema versions catalog. From there, we generate delta code in a common DBMS to reuse all the established features of relational DBMSes.

3 Interfaces of MSVDBs

We now detail the interfaces for logical (Section 3.1) and physical (Section 3.2) management of MSVDBs.

3.1 Logical Version Management

Developers can create, evolve, and drop logical schema versions in the MSVDB. The latter is achieved by executing the following statement:

```
1: DROP VERSION $name_{old}$
```

This removes schema version $name_{old}$ but maintains the table versions if their data is still needed in other versions. A new schema version $name_{new}$ is created from scratch or as an evolution from $name_{old}$ with:

```
1: CREATE VERSION $name_{new}$
   [FROM $name_{old}$] WITH $SMO_1; \dots\ SMO_n;$
```

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. Applications simply request one schema version when connecting to the MSVDB and can then use it like any regular database.

The statement to create and evolve schema versions takes a sequence of **Schema Modification Operations**, already introduced as SMOs, to describe the schema and the data evolution between the versions both forwards and backwards. 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 any number of *outgoing SMOs*. Table 1 summarizes BiDEL's SMOs. The SMOs allow us to create, drop, or rename both tables and columns following the generally known semantics from standard SQL-DDL. Further, we can split and merge tables both vertically and horizontally. 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 can also 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 distributes tuples from 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 unites the tuples of two source table versions into one new target table version.

We formally define a DEL $\mathcal{L}$ to be a set of SMOs with parameters to be instantiated. For instance, the SMO to drop a column requires the name of the table and the name of the column and—for bidirectionality—a function calculating the values for the dropped column in the source version when tuples are inserted in the target version. Let $\mathit{inst}\,(\mathcal{L})$ be the set of all operation instances of $\mathcal{L}$ with valid parameters. Then, a relational database $D = \{R_1, \dots, R_n\}$ with tables R_i can be evolved to another relational database $D' = \{R'_1, \dots, R'_m\}$ at the target side (trg) using the mapping function γ^s_{trg} of an SMO $s \in \mathit{inst}\,(\mathcal{L})$, which is denoted as $D' = \gamma^s_{trg}(D)$. Given a sequence of SMOs $S \in \mathit{inst}\,(\mathcal{L})^+$ with $S = (s_1, s_2, ...s_n)$, a given database D is evolved to another database $D' = \gamma^{s_1}_{trg}(\gamma^{s_2}_{trg}(...\gamma^{s_n}_{trg}(D)))$. In the opposite direction, the mapping γ_{src} is the backward propagation of data to the source side (src).

BiDEL is both practically and relationally complete, so we ensure that developers can intuitively specify any intended evolution without the need to fall back on traditional SQL, which would render the MSVDB inapplicable. Given the evolution history of existing applications, a DEL is **practically complete** when we can model the same evolution exclusively with the given set of SMOs. We validated BiDEL's practical completeness with a benchmark provided by Curino et al. [9] using the evolution history of 171 versions of Wikimedia—the backend of Wikipedia [13,14]. 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.

Therefore, **relational completeness** formally guarantees that any evolution expressible with relational algebra expressions can be expressed with the DEL's SMOs as well. A minimal DEL providing relational completeness is $\mathcal{L}_{\min} = \{\texttt{Add}\,(\cdot,\cdot)\,, \texttt{Del}\,(\cdot)\}$ with

$$\texttt{Add}\,(R', \epsilon) \rightarrow D \cup \{R' = \epsilon\,(R_1, \dots, R_n)\}$$
$$\texttt{Del}\,(R) \rightarrow D \setminus \{R\}$$

The $\texttt{Add}(\cdot,\cdot)$ operation creates a table R' in 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 $S \in \mathit{inst}\,(\mathcal{L}_{\min})^+$, where the tables in D' are computed from D with relational algebra expressions ϵ in the $\texttt{Add}(\cdot,\cdot)$ operation. Thus, $\mathcal{L}_{\min}$ is relationally complete. However, $\mathcal{L}_{\min}$ is not very appealing from a practical standpoint, because it is rather unintuitive and not bidirectional. However, any other DEL that is as expressive as $\mathcal{L}_{\min}$ is relationally complete as well.

We have formally shown the relational completeness of BiDEL [14]. We only sketch the idea here—the important takeaway message is that BiDEL is a feasible DEL for MSVDBs. We defined the semantics of BiDEL's SMOs with relational algebra expressions. Afterwards, we considered every single operation from the relational algebra operations, plus outer joins and the extended projection, and showed that we can provide a semantically equivalent sequence of BiDEL SMOs. While this is trivial for operations such as selection and projection, for instance the cross product requires to temporarily add columns with identical default values and to perform an equality join on those columns. However, BiDEL proved to cover all relational algebra operations. The evolution of further artifacts, such as constraints, is promising future work [7].

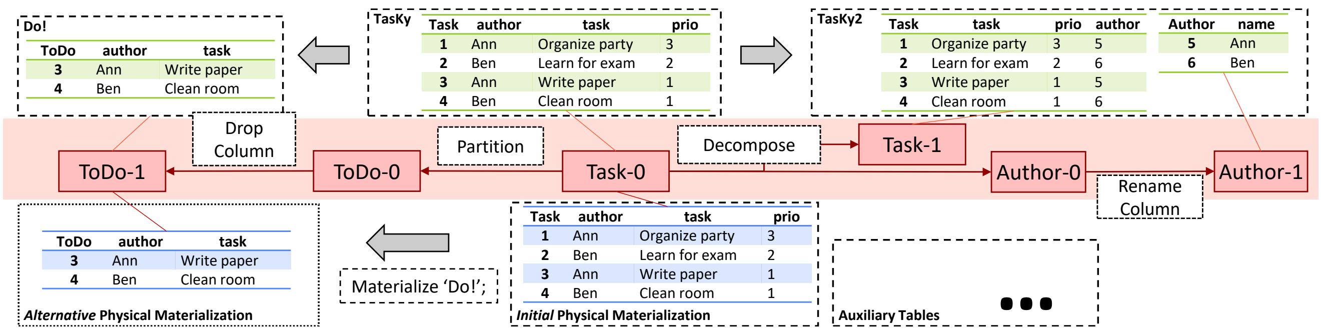
BiDEL's **bidirectionality** is another unique feature and essential for co-existing schema versions in MSVDBs. The arguments of each BiDEL SMO gather enough information to facilitate the automatic generation of delta code for the full propagation of reads and writes between schema versions in both directions. 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. All BiDEL SMOs are designed that way so that developers always specify beforehand how to propagate data both forwards and backwards between schema versions. Table 1 shows the forward semantics with $\mathcal{L}_{\min}$. The inverse SMO then shows the backward semantics; arguments added for bidirectionality are highlighted.

3.2 Physical Version Management

The physical version management interface facilitates easy changes to the physical data representation of the co-existing schema versions. Thanks to the guaranteed **data independence**, the DBA can freely choose to materialize a set of schema versions or a set of table versions. Alternatively, the DBA can ask InVerDa's advisor to propose an optimized materialization and migrate the database accordingly with the click of a button but without any interaction of the developers.

From the view point of a single SMO, the data can be primarily stored in the source, in the target, or in both versions. Initially, data is always materialized in the source schema version. Considering the instantiation of a `PARTITION` SMO, the data is initially stored unpartitioned—the SMO is called *virtualized.* With a migration command, the physical materialization of this SMO can be changed so that the data is physically partitioned—the SMO is then *materialized.* InVerDa also allows redundant materialization with data being stored both unpartitioned at the source side and partitioned at the target side—the SMO is called *redundant.*

Since not all evolutions are information-preserving, InVerDa uses *auxiliary tables* that store the otherwise lost information if the SMO is not redundantly materi-



Do!

ToDo	author	task
3	Ann	Write paper
4	Ben	Clean room

TasKy

Task	author	task	prio
1	Ann	Organize party	3
2	Ben	Learn for exam	2
3	Ann	Write paper	1
4	Ben	Clean room	1

TasKy2

Task	task	prio	author
1	Organize party	3	5
2	Learn for exam	2	6
3	Write paper	1	5
4	Clean room	1	6

Author	name
5	Ann
6	Ben

Alternative Physical Materialization

ToDo	author	task
3	Ann	Write paper
4	Ben	Clean room

Initial Physical Materialization

Task	author	task	prio
1	Ann	Organize party	3
2	Ben	Learn for exam	2
3	Ann	Write paper	1
4	Ben	Clean room	1

Fig. 4: InVerDa architecture with the TasKy example.

alized. These auxiliary tables are initially empty but fed with information when needed to not lose any data—e.g. the values of a dropped column. We exemplarily discuss the auxiliary tables for the PARTITION SMO in Section 5.2. With these auxiliary tables, we ensure that InVerDa can store the data merely in a subset of all table versions without losing any data. Data is propagated through the SMOs from the materialized table versions to the accessed table versions. The generated delta code propagates data accesses either forwards or backwards through the bidirectional SMOs to the closest materialized table versions.

Summary: Developers create and manage logical schema versions while the DBA independently changes their materialization: data independence in MSVDBs.

4 Schema Versions Catalog

The discussed interfaces facilitate the creation and management of schema versions in MSVDBs. The genealogy of all the schema versions and information about their physical materialization are represented in the schema versions catalog. In fact, the catalog is a directed acyclic hypergraph $(\mathcal{TV}, E)$, where each vertex $t \in \mathcal{TV}$ represents a table version and each hyperedge $e \in E$ represents one SMO instance. An SMO instance $e = (S, T)$ evolves a set of source table versions $S \in \mathcal{TV}$ into a set of target table versions $T \in \mathcal{TV}$. 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. At evolution time, InVerDa uses this information to generate delta code that makes all schema versions accessible. At query time, the delta code is executed by the DBMS's query engine—outside InVerDa's components.

Summary & TasKy example: Figure 4 sums up InVerDa's three-layered architecture with the TasKy example. 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 fully updatable views that act like tables in a standard database. On the lower physical layer, a subset of the table versions is physically stored as standard tables in the database. Finally, the schema versions catalog interconnects the materialized table versions and the table versions in the logical schema versions with SMOs. All schema versions can co-exist in the MSVDB and read-/write operations in any schema version are propagated through the bidirectional SMOs to the materialized tables plus auxiliary tables. Written data is 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 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. Assuming 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 migration statement. Now, any data access in the Do! schema version can be directly mapped to the materialized tables which significantly speeds up query processing for this schema version (4 times faster in our evaluation). The price is that the few remaining users of TasKy and especially TasKy2 now face a higher overhead for the additional data propagation.

The bidirectional semantics of BiDEL's SMOs also allow redundant materializations, so we could for instance materialize both schema version Do! and TasKy2 to obtain a high read performance in both schema versions. InVerDa guarantees data independence and the ACID properties on every single schema version, so developers can safely work on one schema version of the MSVDB just as on a regular single-schema database.

5 Guaranteed Data Independence in MSVDBs

MSVDBs, such as InVerDa, guarantee data independence: all schema versions that developers create and manage in the catalog behave like regular single-schema databases no matter how the DBA changes the physical materialization. Co-existing schema versions on one conceptual data set that all behave like regular single-schema databases raise new challenges for the physical design of MSVDBs, as we discuss in Section 5.1. In order to facilitate and safely use the physical design options, we extend the semantics of BiDEL's SMOs to be materialization-independent in Section 5.2. The full formal evaluation is published in a PhD thesis [13]; here, we focus on the conceptual contribution.

5.1 Physical Design

Naïvely, we could materialize each table version. This *fully redundant* approach is the easiest to implement and provides the fastest read access but has the highest space requirements and significant write overhead. The opposite 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, data accesses are also propagated to the auxiliary tables that store the otherwise lost information. Therefore, we extend the bidirectional SMOs to a **materialization-independent mapping semantics** that additionally covers the persistence of auxiliary information in case the respective table versions are not materialized. The materialization-independent mapping semantics easily facilitate any degree of *partially redundant* materializations for an MSVDB where each schema version is guaranteed to behave like a regular single-schema database. In this section, we focus on the non-redundant materialization and show how to formally guarantee the data independence. Non-redundant materialization is the most difficult; 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 tables.

Let us consider a **single SMO**, first. For a single SMO, the envisioned **data independence** means that both the source and the target side of an SMO can be read and written as any common single-schema database. Data written in one schema version should be readable without any changes in the same schema version again, even if the respective schema version is not physically materialized. Figure 5 zooms into the `PARTITION` SMO. Data can be materialized at the source side, at the target side, or at both sides. Processing 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 us start with target-side materialization: The data D_{src} at the source side is 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 [16, 18]—given data at the source side, storing it at the target side, and mapping it back to the source side should return the identical data. For source-side materialization, the same conditions should hold vice versa for target-side data. The data tables that are visible to the user need to survive one round trip—we will use the notation γ^{dat} to project away auxiliary tables. Formally, the following conditions ensure data independence of an SMO:

$$\mathbf{D_{src}} = \gamma^{\mathbf{dat}}_{\mathbf{src}}(\gamma_{\mathbf{trg}}(\mathbf{D_{src}})) \quad (1)$$

$$\mathbf{D_{trg}} = \gamma^{\mathbf{dat}}_{\mathbf{trg}}(\gamma_{\mathbf{src}}(\mathbf{D_{trg}})) \quad (2)$$

The auxiliary tables would be always empty except for SMOs that calculate new values during reads and materialize these values to ensure repeatable reads, e.g., when adding a column, we store the newly calculated values when reading them. We published the formal evaluation of these criteria for BiDEL's SMOs in a PhD thesis [13]. It is safe to say that no data is lost at any side of an SMO, no matter which side is materialize.

The materialized side of an SMO does not necessarily 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 facilitates **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

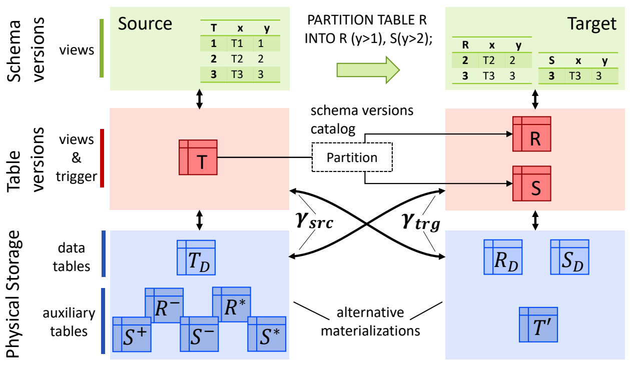


Fig. 5: Mapping functions of single `PARTITION` SMO.

evolution history. All other table versions are provided virtually using the bidirectional and materialization-independent semantics of our BiDEL SMOs. Using results from symmetric relational lenses [16], we extend the guaranteed data independence to chains of SMOs:

$$\mathbf{D_{src}} = \gamma^{\mathbf{dat}}_{\mathbf{1,src}}(\cdots\gamma_{\mathbf{n,src}}(\gamma_{\mathbf{n,trg}}(\cdots\gamma_{\mathbf{1,trg}}(\mathbf{D_{src}})))) \tag{3}$$

$$\mathbf{D_{trg}} = \gamma^{\mathbf{dat}}_{\mathbf{n,trg}}(\cdots\gamma_{\mathbf{1,trg}}(\gamma_{\mathbf{1,src}}(\cdots\gamma_{\mathbf{n,src}}(\mathbf{D_{trg}})))) \tag{4}$$

Hence, the data is materialized anywhere in the schema evolution history and each schema version is guaranteed to behave like a regular single-schema database. BiDEL, which satisfies the data independence criteria, guarantees developers that they can safely create and use new schema versions and the DBA can safely change the materialization without any risk of losing data. We guarantee that users can use a single schema version just like a regular database and changes are propagated to other versions with a best-effort strategy; if users use multiple versions, we need to carefully design the application to not confuse the users.

5.2 Materialization-Independent Mapping Semantics

We utilize BiDEL's bidirectionality to propagate data both forwards and backwards between table versions—not all of them are necessarily materialized. The mapping semantics needs to be materialization-independent o not lose any information. We exemplarily discuss the PARTITION SMO. The remaining SMOs are covered in [13, 15]. In Figure 5, a source table T is horizontally partitioned into two target tables R and S based on the incomplete and overlapping conditions c_R:(y>1) and c_S:(y>2). 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.

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. With target-side materialization, 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. The payload data of R, S, and T is stored in the physical tables R_D, S_D, and T_D. The auxiliary tables R^-, S^+, S^-, R^*, S^*, and T' store all otherwise lost information. E.g., T' stores all tuples that do not match any of the partition criteria—we detail on all auxiliary tables later in this section. There are different ways of defining γ_{trg} and γ_{src}; our proposal 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. There are many possible structures for the auxiliary information, but at the end of the day they all have to store the same minimal set of 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. Thus, the predicates $T(p, \_)$ and $\neg T(p, \_)$ are always unambiguous and the multiset semantics of a relational database fits with the set semantics of Datalog, as the unique key p prevents equal tuples in a relation.

We start with the γ_{trg} mapping of a **materialized** PARTITION SMO that horizontally splits a table T into two tables R and S on conditions c_R and c_S (Figure 5):

$$R(p, A) \leftarrow T(p, A), c_R(A) \tag{5}$$

$$S(p, A) \leftarrow T(p, A), c_S(A) \tag{6}$$

This is not sufficient for the desired materialization-independent semantics, as T may contain tuples neither captured by c_R nor by c_S. To avoid information loss, we store the uncovered tuples with $c_S(A) \vee c_R(A) = \perp$ in the auxiliary table T' on the target side:

$$T'(p, A) \leftarrow T(p, A), \neg c_R(A), \neg c_S(A) \tag{7}$$

When we materialize the PARTITION SMO, we only store R and S as well as the auxiliary table T' but not T.

Let us now consider the γ_{src} mapping function for reconstructing T from the target side, which is essentially a union of R, S, and T'. Since c_R and c_S are not necessarily disjoint, there are tuples with $c_S(A) \wedge c_R(A) = \top$. Source tuples that occur as two equal but independent instances in R and S are called *twins*. Independently updated twins result in *separated twins*. To resolve this ambiguity and ensure data independence, we consider the first twin in R to be the preferred twin and define γ_{src} of PARTITION to return all tuples in R as well as those tuples in S that are not contained in R:

$$T(p, A) \leftarrow R(p, A) \tag{8}$$

$$T(p, A) \leftarrow S(p, A), \neg R(p, \_) \tag{9}$$

$$T(p, A) \leftarrow T'(p, A) \tag{10}$$

The Rules 5–10 define sufficient semantics for `PARTITION` as long as the target side is materialized.

Now, consider the `PARTITION` SMO to be **virtualized**. R and S can still contain separated twins. According to Rule 9, T contains the separated twin from R. To avoid losing the other twin, it is stored in the auxiliary table S^+:

$$S^+(p, A) \leftarrow S(p, A), R(p, A'), A \neq A' \tag{11}$$

Accordingly, γ_{trg} reconstructs the separated twin in S from S^+ instead of T (concerns Rule 6). Twins can also be deleted independently resulting in a *lost twin*. 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^-:

$$R^-(p) \leftarrow S(p, A), \neg R(p, \_), c_R(A) \tag{12}$$
$$S^-(p) \leftarrow R(p, A), \neg S(p, \_), c_S(A) \tag{13}$$

Then, γ_{trg} uses this information to explicitly exclude those tuples (concerns Rules 5 and 6): When tuples are updated in R or S and do not meet the conditions c_R or c_S any longer, auxiliary tables R^* and S^* are employed for identifying those tuples to include them in γ_{trg} (concerns Rules 5 and 6).

$$S^*(p) \leftarrow S(p, A), \neg c_S(A) \tag{14}$$
$$R^*(p) \leftarrow R(p, A), \neg c_R(A) \tag{15}$$

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

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

$$R(p, A) \leftarrow T(p, A), c_R(A), \neg R^-(p) \tag{16}$$
$$R(p, A) \leftarrow T(p, A), R^*(p) \tag{17}$$
$$S(p, A) \leftarrow T(p, A), c_S(A), \neg S^-(p), \neg S^+(p, \_) \tag{18}$$
$$S(p, A) \leftarrow S^+(p, A) \tag{19}$$
$$S(p, A) \leftarrow T(p, A), S^*(p), \neg S^+(p, \_) \tag{20}$$
$$T'(p, A) \leftarrow T(p, A), \neg c_R(A), \neg c_S(A), \neg R^*(p), \neg S^*(p) \tag{21}$$

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

$$T(p, A) \leftarrow R(p, A) \tag{22}$$
$$T(p, A) \leftarrow S(p, A), \neg R(p, \_) \tag{23}$$
$$T(p, A) \leftarrow T'(p, A) \tag{24}$$
$$R^-(p) \leftarrow S(p, A), \neg R(p, \_), c_R(A) \tag{25}$$
$$R^*(p) \leftarrow R(p, A), \neg c_R(A) \tag{26}$$
$$S^+(p, A) \leftarrow S(p, A), R(p, A'), A \neq A' \tag{27}$$
$$S^-(p) \leftarrow R(p, A), \neg S(p, \_), c_S(A) \tag{28}$$
$$S^*(p) \leftarrow S(p, A), \neg c_S(A) \tag{29}$$

In sum, these bidirectional mapping semantics of the `PARTITION` SMO allow us 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—we formally guarantee data independence (Conditions 1 and 2) for all those BiDEL SMOs [15,13]. To this end, we calculated and reduced the semantics for subsequently applying γ_{src} and γ_{trg}, which should result in an identity mapping. Thus, data from one side of an SMO that is persisted at the other side of the SMO can always be read again at the initial side without any information loss.

TasKy example: We now apply the materialization-independent semantics in our `TasKy` example. When we create the partition `Todo` from the initial table `Task`, we essentially set $c_R = c_{todo} : (prio = 1)$ and $c_S : \perp$. As a result, the literals c_S and $S(p, A)$ will never be satisfied, which allows us to ignore the second partition and reduce the rule set:

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

$$\texttt{Todo}(p, A) \leftarrow \texttt{Task}(p, A), c_{todo}(A) \tag{30}$$
$$\texttt{Todo}(p, A) \leftarrow \texttt{Task}(p, A), \texttt{Todo}^*(p) \tag{31}$$
$$\texttt{Task}'(p, A) \leftarrow \texttt{Task}(p, A), \neg c_{todo}(A), \neg \texttt{Todo}^*(p) \tag{32}$$

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

$$\texttt{Task}(p, A) \leftarrow \texttt{Todo}(p, A) \tag{33}$$
$$\texttt{Task}(p, A) \leftarrow \texttt{Task}'(p, A) \tag{34}$$
$$\texttt{Todo}^*(p) \leftarrow \texttt{Todo}(p, A), \neg c_{todo}(A) \tag{35}$$

Intuitively speaking, the target side table version `Todo` contains all the tuples that satisfy the condition c_{todo} (Rule 30) while the auxiliary table `Task`$'$ contains all remaining tuples (Rule 32). Then, the source-side table version `Task` is essentially the union of these two tables (Rules 33 and 34). For source-side materialization, the auxiliary table `Todo`* stores the identifiers of all tasks that should occur in `Todo` at the target side but that have been updated and do not meet the condition c_{todo} any longer (Rule 35). We use this auxiliary table in the γ_{trg} mapping to add those tuples to `Todo` that have their identifier in `Todo`* (Rule 31) and consequently exclude those tasks from the auxiliary table `Task`' (Rule 32).

To show the whole power of the `PARTITION` SMO, consider the partition criteria $c_R = c_{todo} : (prio = 1)$ and additionally $c_S = c_{should\_do} : (prio <= 2)$ for the second partition `should_do`. The tasks with $prio = 2$ occur in both partitions, while the tasks with $prio = 3$ are not represented at all. We do need the other auxiliary tables to persist data at the unpartitioned source side, now. Namely, `should_do`$^*(p)$ keeps the identifiers of all tasks that belong to `should_do` even though they are updated to not meet the condition any longer. Further, those tuples with $prio = 2$ are twins that occur in both partitions. If the twins are updated independently in `Todo` and `should_do`, then the auxiliary table $\texttt{should\_do}^{+}(p, A)$ holds the second twin (from `should_do`) while the first (from `Todo`) is stored in `Task`. If one tuple is deleted in one twin but remains in the other twin, then the auxiliary tables $\texttt{Todo}^{-}(p)$ and $\texttt{should\_do}^{-}(p)$ hold the respective identifiers to make sure that the tuples are not added again at the target side.

Summary: We ensure data independence for any evolution with sequences of SMOs. No matter which table versions we materialize, InVerDa manages auxiliary information to guarantee that any schema version in the MSVDB behaves like a regular single-schema database. To our best knowledge, BiDEL is the first set of powerful SMOs with validated data independence.

6 Delta Code Generation

InVerDa realizes data independence in MSVDBs by generating delta code for all table versions that are not physically materialized. We translate the γ_{src} and γ_{trg} mappings of those SMOs that connect unmaterialized table versions with the materialized ones into SQL view and trigger definitions—details can be found in [15]. The delta code for a specific table version depends on the materialization state of the table's adjacent SMOs, i.e., 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 materializations, consider the exemplary evolution in Figure 6. Schema version T_i is materialized, hence the SMO instance i stores its data at the target side and is materialized. The SMO instance $i-1$ is also materialized, as we use the auxiliary tables to persist data at the target side—however, the data tables T_{i-1} do not exist physically but are virtually provided by SMO i. Thanks to the guaranteed data independence of BiDEL's SMOs the data access propagation can be safely cascaded like this. Analogously, 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

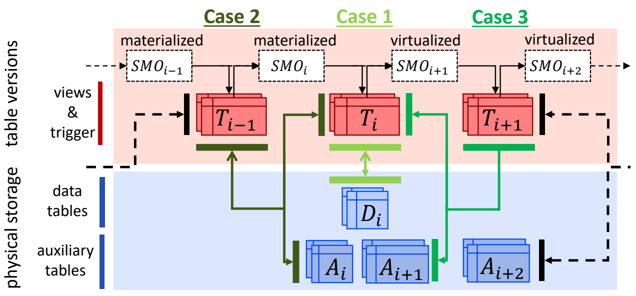


Fig. 6: Three different cases in delta code generation.

be distinguished, depending on the direction a specific table version needs to go to reach the materialized data.

Case 1 – local: The incoming SMO is materialized; all outgoing SMOs are virtualized. The data T_i is directly stored and accessed in data table D_i.
Case 2 – forwards: The incoming SMO and one outgoing SMO are materialized. The data of T_{i-1} is stored in newer table versions, so data access is propagated with γ_{src} (read) and γ_{trg} (write) of $\texttt{SMO}_i$.
Case 3 – backwards: The incoming SMO and all outgoing SMOs are virtualized. The data of T_{i+1} is stored in older table versions, so data access is propagated with γ_{trg} (read) and γ_{src} (write) of $\texttt{SMO}_{i+1}$.

In Case 1, delta code generation is trivial. In Case 2 and 3, InVerDa translates the Datalog rules into views for reading and instead-of triggers on those views for writing. For physical migrating, InVerDa generates queries for all required data and auxiliary tables to materialize them and updates the delta code of all affected table versions. Datalog's expressiveness is a subset of SQL and the translation is straightforward [13,15].

TasKy example: Assume the alternative materialization as shown in Figure 4 where the Do! schema version is physically materialized. Then both the `DROP COLUMN` and the `PARTITION` SMO are materialized while the `DECOMPOSE` and the `RENAME COLUMN` SMO are virtualized. Hence, InVerDa creates the table `Todo-1` physically and generates delta code for all other table versions. There is a view for the table version `Todo-0` that joins the values for the dropped column from an auxiliary table—this view is made updatable with a trigger that updates both the data table `Todo-1` and the auxiliary table with the dropped column. The view for the initial `Task-0` table version then merges the data from the view `Todo-0` with an auxiliary table that holds all eliminated tasks. Again, this view is made updatable with a trigger that updates data in the auxiliary table or propagates it further to view `Todo-0`. In the same spirit, InVerDa generates updatable views for the remaining table versions to make all three schema versions Do!, TasKy, and TasKy2 fully accessible.

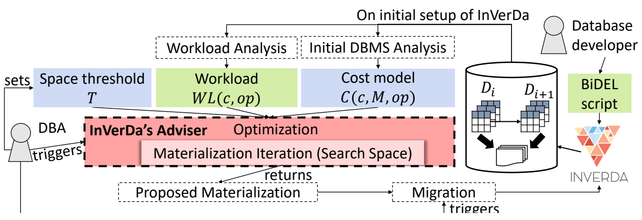


Fig. 7: Workflow of InVerDa's advisor.

Summary: 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 materialized table versions. Data accesses are propagated step-by-step with standard database objects (views/triggers), which keeps the delta code simple and preserves performance and transaction guarantees of common database systems also for MSVDBs.

7 Physical Design Advisor for MSVDBs

Co-existing schema versions in an MSVDB are different representations of the same conceptual data set. Thanks to the data independence of BiDEL's SMOs, there is no need to materialize each and every schema version to let them co-exist, but the DBA can freely choose a subset of all table versions to materialize, which opens up new **physical design** opportunities. The search space of possible materializations can grow exponentially with the number of table versions and is hard to explore, hence we equip the DBA with an advisor. Therefore, we discuss its objective in Section 7.1. The advisor is based on a cost model that we introduce in Section 7.2. We define the search space in Section 7.3 and present the optimization algorithm in Section 7.4.

7.1 Objective

The advisor covers the whole search space from non-redundant over partially redundant to fully redundant materializations. In a non-redundant materialization, only one preferred schema version is materialized; accesses to all other versions are propagated to this preferred version and to the auxiliary tables. In contrast, a fully redundant materialization physically stores every single table version requiring way more storage space. However, no auxiliary tables are needed, which speeds up query processing in turn. Partially redundant materializations strike a balance between the two extremes by materializing a subset of the table versions.

InVerDa's advisor follows the workflow shown in Figure 7. Triggered by the DBA, it determines the set of materialized table versions that provides the best overall performance for the current workload under a given space threshold. The advisor is cost model-based and uses an evolutionary optimization algorithm. When the DBA confirms the proposal, InVerDa migrates the data and all schema versions continuously work, but now provide in total a higher performance.

We will discuss all artifacts of InVerDa's advisor workflow to precisely specify the actual optimization problem. To this end, we introduce a formal notation: Let $\mathcal{SV}$ be the set of all schema versions in the MSVDB. Then $\mathcal{TV}(s)$ is the set of table versions in schema $s \in \mathcal{SV}$, and $\mathcal{TV}$ is the set of all table versions in the MSVDB. Let $\mathcal{O} = \{\text{select}, \text{insert}, \text{update}, \text{delete}\}$ be the set of operations to be executed on any table version $t \in \mathcal{TV}$. Let $\mathcal{MATS} \subseteq \{\mathcal{M} | \mathcal{M} \subseteq \mathcal{TV}\}$ be the set of all valid materialization states that materialize enough table versions to enable accesses to any table version. Then, $\mathcal{M}$ is a set of table versions that is physically materialized, $\mathcal{S}(\mathcal{M})$ is its size in bytes, and $\mathcal{MATS}(\mathcal{T})$ is the subset of $\mathcal{MATS}$ that meets the space constraint $\mathcal{T}$. The function $\mathcal{E}(t)$ returns the set of all directed hyper edges (SMOs) that are connected to the given table version $t \in \mathcal{TV}$. Finally, $\mathcal{N}(e, t)$ returns the set of all table versions that are at the opposite end of the directed hyper edge e (SMO) of the given table version $t \in \mathcal{TV}$.

Workload description: The workload is defined as a function $\mathcal{WL}(t, op)$, which returns the percentage of a given operation $op \in \mathcal{O}$ on a given table version $t \in \mathcal{TV}$ w.r.t. the whole workload:

$$\mathcal{WL} : \mathcal{TV} \times \mathcal{O} \rightarrow [0; 1] \tag{36}$$

All table versions $t \notin \{t | t \in \mathcal{TV}(v), v \in \mathcal{SV}\}$ that are not part of any schema version are never explicitly accessed, hence their workload share is zero. The sum of all operations on all table versions is 100%.

Learned cost model: Propagating read and write accesses through SMOs naturally causes costs that need to be minimized by InVerDa's advisor. Temporarily changing the materialization to actually measure the costs can easily take several minutes to multiple days in realistic scenarios. Instead, the advisor uses a cost model to quickly estimate the costs. We define the cost model with the function $\mathcal{C}(t, \mathcal{M}, op)$, 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 $t \in \mathcal{TV}$:

$$\mathcal{C} : \mathcal{TV} \times \mathcal{MATS} \times \mathcal{O} \rightarrow \mathbb{R}^+ \tag{37}$$

Since the used hardware and DBMS influence the actual costs for access propagating, the cost function is individually *learned* when setting up InVerDa.

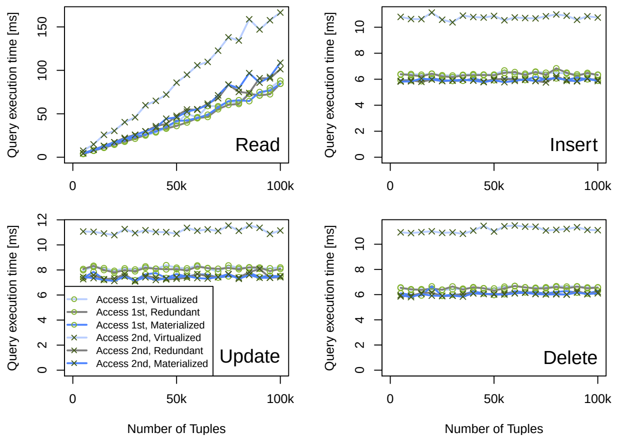


Fig. 8: Accesses propagation through `ADD COLUMN` SMO.

Optimization objective: Putting everything together, the optimization objective of InVerDa's advisor is to find a set of materialized table versions $\mathcal{M} \in \mathcal{MATS}$ with its size being within the threshold $\mathcal{S}(\mathcal{M})$ that provides the minimal costs for accessing the data. The costs are calculated as the sum of all operations *op* on all table versions t, weighted with their share of the overall workload. The overall objective is to find the materialization $\mathcal{M}_{best} \in \mathcal{MATS}(\mathcal{T})$ with:

$$\underset{\mathcal{M} \in \mathcal{MATS}(\mathcal{T})}{arg\ min} \left(\sum_{t \in \mathcal{TV}} \sum_{op \in \mathcal{O}} \mathcal{WL}(t, op) \cdot \mathcal{C}(t, \mathcal{M}, op) \right) \quad (38)$$

Intuitively, we minimize the expected execution time for the workload.

7.2 Cost Model

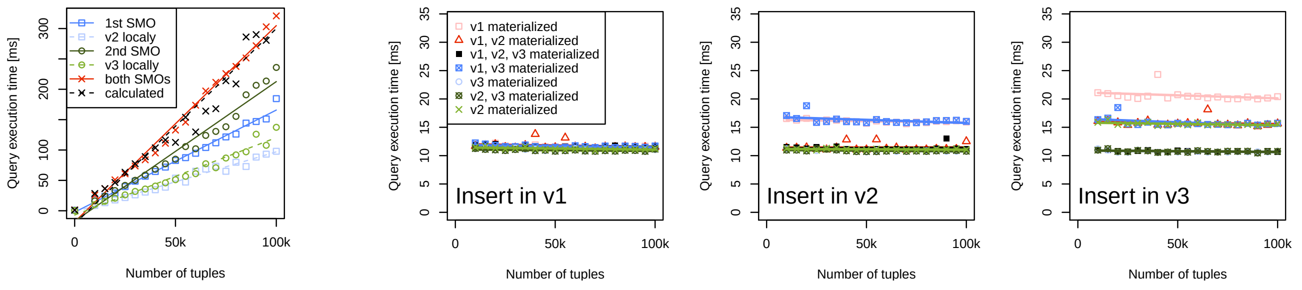
The costs depend on the generated delta code, the used DBMS, the used hardware, and so on. Hence, we learn the specific cost function *for each system individually* by running sample workloads on sample data. In the following, we analyze simple scenarios to learn the general characteristics that make up the cost model with parameters to be trained for a specific MSVDB.

Performance of single SMOs: Most SMOs have an **asymmetric overhead** w.r.t. the materialization, which means that the propagation in one direction is more expensive than in the other direction. For each operation $op \in \mathcal{O}$ on each SMO, we consider six different scenarios: The workload accesses either the source (Access 1st) or the target versions (Access 2nd), while 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). Figure 8 shows the query execution for a very simple scenario: A table with 2 columns and a growing number of 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. Whenever we access the target side of the virtualized SMO, we face a significantly higher overhead than in all other cases, because the data is primarily stored without the added column, hence reading or writing the new version requires to join with or write to the auxiliary table that persists the new column. Reading data in the opposite direction is done by merely projecting away the new column, explaining the significant asymmetry of the `ADD COLUMN` SMO. So, the asymmetry of the `ADD COLUMN` SMO suggests to materialize or replicate it, as long as space constraints and neighboring SMOs do allow this. Further, the costs for accessing data locally are always identical no matter whether the respective other side of the SMO is materialized as well or not. Due to this observation, we define the cost model along the two dimensions: (1) access either at target or at source side and (2) access either locally or remotely with auxiliary tables. Hence, we need to learn cost function for only four different scenarios.

Finally, we see that read accesses scale linearly with the number of tuples, while writing requires almost constant time. As a consequence, we can use functions of the form $ax+b$ with x being the source tables' size to describe the cost function for access propagation through single SMOs. For writing through an `ADD COLUMN` SMO, a constant function would be enough, however, this does not apply to all SMOs, which motivates using 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, although the presented concepts easily support the extension to higher order polynomials or logarithmic functions if needed.

We used the `ADD COLUMN` SMO without loss of generality. The measurements for the remaining SMOs are published in the PhD thesis [13] and show similar characteristics. The different asymmetries of the different SMOs are not intuitively obvious for DBAs, as DBAs are not necessarily aware of the managed auxiliary tables and the delta code generation that significantly affect the actual overhead. Our cost model-based advisor hides all the complexity and releases DBAs from the need to understand the asymmetries in the first place.

Performance of SMO sequences: For the access propagation through multiple SMOs, the overheads of the single SMOs **combine linearly**. We conduct a mi-



(a) Read operation. (b) Write operations.

Fig. 9: Scaling behavior for access propagation through two subsequent `ADD COLUMN` SMOs.

cro benchmark on all possible evolutions with two subsequent SMOs and three resulting 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. Again, we focus on two subsequent `ADD COLUMN` SMOs and point the interested reader to [13] for the remaining combinations.

For **reading**, 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 9(a) 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.

Conducting the same analysis for **writing** reveals that the overhead also combines linearly which facilitates stable estimations of the actual costs. Figure 9(b) shows the execution time of insert operations in three exemplary schema versions that are connected by two `ADD COLUMN` SMOs. 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 values of the two added columns are stored in auxiliary tables, which increases the required time. For writing at the third schema version with only the first version being materialized, we face this overhead twice; 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.

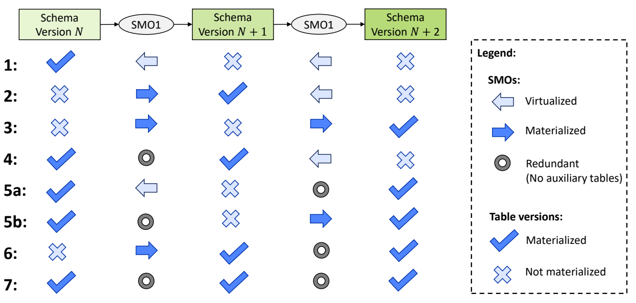
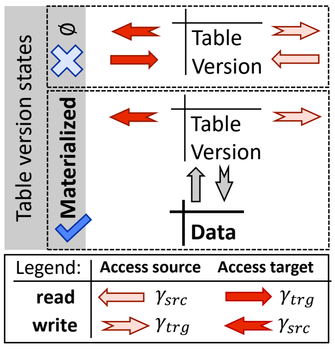


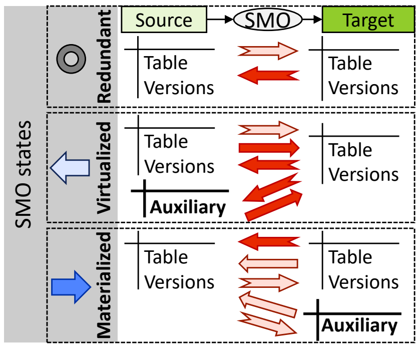
Fig. 10: Materialization states of two subsequent SMOs.

We conducted the same measurements for all combinations of SMOs and empirically confirmed that the execution time of read and write operations through sequences of SMOs is **easily computable** [13]. It is the **linear combination** of the respective execution times of the propagation through single SMOs. So, we build a stable cost model for evolutions based on cost models for single SMOs. Reading along SMOs usually requires joins with auxiliary tables and writing requires a stored procedure call. Both things are independent from other SMOs and explain the linear combination of the costs. In the evaluation, we show that this assumption is stable enough to obtain very precise estimations.

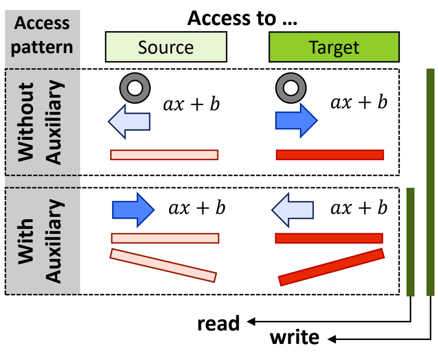
Data access patterns: Figure 10 shows all possible materialization states for two subsequent SMOs: each schema version can be materialized or not—at least one schema version must be materialized in order to not lose all the data. The set of materialized table versions determines the materialization state of the SMOs, which in turn determine the required auxiliary tables and the delta code generation as discussed in Section 6. There is one ambiguity: In Lines 5a and 5b, both the first and the last schema version are materialized so 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 exclusively to 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.



(a) Table version states.



(b) SMO states.



(c) Cost model for access patterns.

Fig. 11: Access patterns for different materialization states.

In Figure 11, we analyze the different possible access patterns determined by the materialization states of the table versions. Figure 11(a) shows that a single table version can be either materialized or not. The materialization states of the table versions determine the materialization states of the SMOs, as shown in Figure 11(b). If both the source and the target table version are materialized, the SMO is called redundant and no auxiliary tables are needed—no information will be lost anyhow. Merely write operations need to be propagated through redundant SMOs to ensure global consistency among all table versions. If the data is stored only at the source side, the SMO is called virtualized. Then, read-/write operations on the target side are answered by accessing both the source table versions and the auxiliary tables. 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. Please note that the source table versions are not necessarily materialized, but could be provided virtually by another neighboring SMO. For a materialized SMO, the same patterns apply vice versa.

Learned cost model: The analysis showed that the costs for subsequent SMOs are the **linear combination** of the single SMOs—the query execution time grows monotonously in chains of SMOs and **scales linearly** with the number of tuples. We now condense this knowledge into a *cost model* to estimate the costs for executing a given operation on a given table version with a given materialization schema. These costs are mainly determined by the data access pattern and the involved auxiliary tables. Summarizing the possible access patterns from Figure 11(b), we see that there are four different cases as shown in Figure 11(c): (1) forwards without auxiliary tables, (2) backwards without auxiliary tables, (3) forwards with auxiliary tables, and (4) backwards with auxiliary tables. For each case, we learn 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. In order to learn the cost model's parameters, we create exemplary scenarios with one SMO respectively and a growing number of synthetic tuples as in the previous measurements. We then measure the propagation overhead and use linear regression to learn the parameters a and b. Further, we learn a linear cost model for reading and writing any materialized table version locally in the same manner.

The cost function $\mathcal{C}(t, \mathcal{M}, op)$ returns the estimated costs for executing the operation op on the table version t with a given set of materialized table versions $\mathcal{M}$. We estimate the costs in two steps. First, we derive the SMOs' materialization states from the given set of materialized table versions $\mathcal{M}$. This step contains a nested optimization problem: for each table version, we determine the cost-wise closest materialized table versions. 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, so there is no cyclic dependency between the shortest path search and the overall cost estimation. Second, we estimate and sum up the costs for each operation op on each table version t weighted according to the workload $\mathcal{WL}(t, op)$. Read operations are directed to materialized table versions on the cheapest path. To prevent information loss, write operations are propagated through the whole schema versions catalog since auxiliary tables might be affected anywhere.

In summary, we used the detailed analysis of the data access propagation to define a cost model that precisely estimates 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 (Section 8.4). Most importantly, the learned cost model estimates the costs orders of magnitude faster than an actual measurement on a physically migrated database.

7.3 Search Space

To find the best materialization, we have to know the whole search space in the first place. Intuitively, every single table version in the evolution history can be either materialized or not, which yields $2^{|\mathcal{TV}|}$ possible materializations—however, not all of these materializations are valid. When, e.g., not a single table version is materialized, then the data will be lost.

A **valid materialization** ensures that each and every table version $t \in \mathcal{TV}$ can propagate its data access to materialized table versions. The respective table version t is called **covered** then. A table version $t \in \mathcal{TV}$ is covered by a materialization $\mathcal{M}$, so $cov(t, \mathcal{M})$ holds, when there exists a hyper path through the schema versions catalog that ends at materialized table versions. It can be a hyper path, because SMOs with two source or two target tables may distribute the data to both of them, so they both need to be covered.

$$cov(t, \mathcal{M}) \equiv t \in \mathcal{M} \vee \left(\bigvee_{e \in \mathcal{E}(t)} \left(\bigwedge_{t' \in \mathcal{N}(e,t)} cov(t') \right) \right) \tag{39}$$

This ensures that the data of t is stored at least once. A materialization $\mathcal{M}$ is valid when all table versions are covered. The set of all valid materialization states is defined as:

$$\mathcal{MATS} \equiv \{\mathcal{M} | \mathcal{M} \subseteq \mathcal{TV} \wedge \forall_{t \in \mathcal{TV}}\ cov(t, \mathcal{M})\} \tag{40}$$

The advisor may only choose those materializations that meet the given space threshold $\mathcal{T}$:

$$\mathcal{MATS}(\mathcal{T}) \equiv \{\mathcal{M} | \mathcal{M} \in \mathcal{MATS} \wedge \mathcal{S}(\mathcal{M}) \leq \mathcal{T}\} \tag{41}$$

This resulting **search space** can **grow exponentially** with the number of table versions, which is determined by the catalog's graph structure. In an MSVDB with N created table versions, which are never evolved, there is only one valid materialization with all tables. If all N table versions are evolved with one SMO, we have N independent evolutions and $|\mathcal{TV}| = 2N$ 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 long evolution with M SMOs that e.g. add columns to one table, facilitate $2^{M+1} - 1$ possible materializations, as any non-empty subset of all table versions can be materialized. Hence, both multiple independent evolutions as well as sequences of dependent evolutions cause an exponential search space size. To provide a generally applicable advisor, we tailor the optimization algorithm to a potentially exponentially large search space in the next subsection.

7.4 Optimization Algorithm

Following the definition of $\mathcal{MATS}$, there are 59 possible materializations for the six table versions in TasKy, and there are up to 1×10^{61} possibilities for the 203 table versions in the Wikimedia Benchmark [9]. This makes it impracticable to enumerate the whole search space. Instead, we 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. The cost function is only defined globally for a complete materialization and cannot assign local costs for materializing one specific table version. 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.

Let the costs for not materializing a specific table version t be very high because data is propagated through an expensive neighboring SMO. These costs may be reduced to almost zero if we merely materialize another table version t' that was, e.g., created by renaming a column in table version t, so the costs for locally materializing single table versions are not independent from each other. Therefore, the optimizer's objective 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 [11].

Further, when enumerating the search space by stepwise extensions of the materialization with 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 $\mathcal{MATS}$. We can neither assume that $\mathcal{C}(t, \mathcal{M}_1, op) \leq \mathcal{C}(t, \mathcal{M}_1 \cup \mathcal{M}_2, op)$ holds nor can we assume $\mathcal{C}(t, \mathcal{M}_1, op) \geq \mathcal{C}(t, \mathcal{M}_1 \cup \mathcal{M}_2, op)$. As 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.

In summary, the global cost function and the non-monotone behavior of extending materializations prohibit to divide the optimization problem into smaller subproblems. Hence, dynamic programming algorithms are not an option. Also, greedy algorithms return solutions that are far from optimal as we will see in the evaluation. **Evolutionary optimization algorithms** are currently the best approach for such problems [11]. They maintain a population of the currently best solutions and randomly evolve them by small manipulations but also by more significant mutations [4]. Following Darwin's Law, only the best solutions survive and are evolved further to continuously improve the solutions.

The evolutionary optimization algorithm creates an **initial population** with non-redundant materialization of the youngest and oldest table versions, partly redundant materializations with greedily chosen schema versions, as well as a fully redundant materialization. As **evolution steps**, we use fine-grained fuzzy modifications that move, 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 state. Our cost function efficiently estimates the costs for a given materialization; the evolutionary optimization algorithm removes those materializations with the highest estimated costs from our population and continues the optimization until it reaches a given maximum number of considered materializations or does not see any improvement for a certain time.

Summary: InVerDa's advisor optimizes the materialization using an evolutionary algorithm to handle the non-monotone cost function. It relies on an individually learned cost function to quickly evaluate potential materializations. In the evaluation, we will show that this evolutionary algorithm finds the optimal solution for small examples such as our TasKy example and that it achieves significant improvements for long evolution histories such as the evolution of Wikimedia.

8 Evaluation

InVerDa allows applications to continuously access all co-existing schema versions, while developers can focus on the continuous implementation of the software without caring about former versions. The DBA can change the materialization with the click of a button and especially without restricting the availability of the co-existing schema versions or invalidating the developers' code. We have formally shown that all schema versions in an MSVDB behave like regular single-schema databases and developers and DBAs can safely use the provided features without risking inconsistencies or information loss. Now, we evaluate our prototypical implementation. InVerDa automatically generates the delta code based on the discussed Datalog rules; we analyze the gain in simplicity and robustness of database development (Section 8.1) and measure the overhead of InVerDa's delta code (Section 8.2) in comparison to a handwritten SQL implementation of co-existing schema versions. Further, we evaluate the optimization potential of InVerDa's flexible materialization (Section 8.3) and empirically analyze InVerDa's advisor focusing both on the cost model (Section 8.4) and on the actual optimization algorithm (Section 8.5).

InVerDa	Initially	Evolution	Migration
LOC	1	3	1
Stmts	1	3	1
Chars	54	152	19
SQL (Ratio)			
LOC	1 (×1.00)	359 (×119.67)	182 (×182.00)
Stmts	1 (×1.00)	148 (×49.33)	79 (×79.00)
Chars	54 (×1.00)	9477 (×62.35)	4229 (×222.58)

Table 2: Ratios between SQL and InVerDa delta code.

Setup: We evaluate two different scenarios: (1) our small and comprehensive TasKy example and (2) 171 versions of Wikimedia [9] as a real-world scenario. We measure single thread performance in a PostgreSQL 9.4 database on a Core i7 machine with 2,4GHz and 8GB. We load Wikimedia with Akan Wiki in the 109th version *v16524* with 14 359 pages and 536 283 links.

8.1 Simplicity

TasKy: BiDEL releases developers from the expensive and error-prone manual implementation of delta code. We implement the evolution from TasKy to TasKy2 with handwritten and hand-optimized SQL and compare this code to the semantically equivalent BiDEL statements. We manually implement (1) creating the initial TasKy schema, (2) creating the new co-existing schema version TasKy2 with the respective views and triggers, and (3) migrating the physical materialization to TasKy2 and adapting all existing delta code. This handwritten SQL code is much longer and much more complex than doing the same with BiDEL. Table 2 shows the lines of code (LOC) required with SQL and BiDEL, respectively, as well as the ratio between these values. For comparison, we include the number of statements and number of characters (consecutive white-space characters counted as one) to get an objective picture. 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 of SQL code, respectively. We can express the same with 3 and 1 lines with BiDEL, respectively. Moreover, the SQL code is also more complex, as indicated by the average number of characters per statement. BiDEL is working exclusively on the visible schema versions. It is literally impossible to corrupt the data with BiDEL, since we guarantee that existing schema versions are not affected by evolutions. With handwritten SQL, developers also have to manage auxiliary tables, triggers, etc.—manually working on multiple technical levels also increases the code's complexity and is error-prone and expensive.

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

Table 3: SMOs of Wikimedia schema evolution.

Wikimedia: Curino et al. defined a database evolution benchmark with 171 schema versions of Wikimedia [9]. A practically complete DEL should allow us to model this evolution history of Wikimedia only with the given SMOs. In fact, BiDEL is capable of doing this with 209 SMOs. Table 3 shows the number of occurrences of each SMO (Nof) in this evolution. 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, which confirms the need for more sophisticated database evolution support. In general, the evolution with BiDEL is very similar to the same evolution modeled with PRISM as used in the original benchmark [9]—however, the sets of SMOs differ slightly. For instance, the evolution from version v06696 to v06710 takes 31 BiDEL SMOs instead of 92 PRISM SMOs due BiDEL's powerful DECOMPOSE SMO. In sum, BiDEL is relationally and practically complete and it is orders of magnitude shorter and more robust than common SQL.

8.2 Overhead of Generated Delta Code

We rely on the database optimizer to find a fast execution plan for InVerDa-generated delta code. We will now show that the overhead compared to hand-optimized SQL is reasonably small.

TasKy: As before, we use the handwritten SQL for the evolution from TasKy to TasKy2 and compare its performance to the semantically equivalent evolution with BiDEL. Both TasKy and TasKy2 co-exist and the physical materialization will be migrated from TasKy to TasKy2 eventually. The automated delta code generation does not only eliminate the error-prone and expensive manual implementation, but it is also reasonably fast. Creating the initial schema version and making it available as a new schema version for applications took 154 ms. The evolution to TasKy2 with two SMOs requires 230 ms for both the generation and execution of the evolution script. The same took 177 ms for Do!.

In Figure 12, we feed the TasKy and TasKy2 schema versions with 100 000 tasks and compare the performance of the 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 100 tuples to TasKy, and (4) for inserting 100 tuples to TasKy2. We evaluate both the initial materialization according to TasKy and the evolved materialization according to TasKy2. The generated delta code causes very little overhead (up to 4 %). Keeping in mind the difference in length and complexity of the code (182 x LOC for the migration), an overhead of 4 % on average is more than reasonable for most users. Further, 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 inserting new tasks, the overhead is just as small—interestingly, the evolved materialization is always faster for writing because the initial materialization requires an additional auxiliary table for the foreign key relationship. In sum, the price for agile database evolution with automated delta code generation is only a quite small overhead.

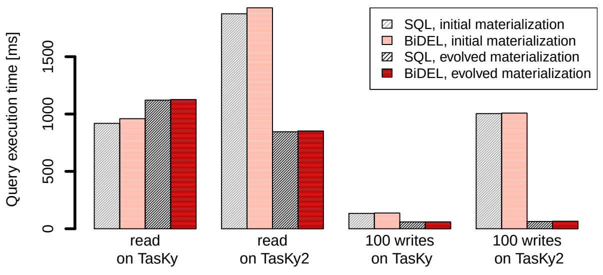


Fig. 12: Overhead of generated code.

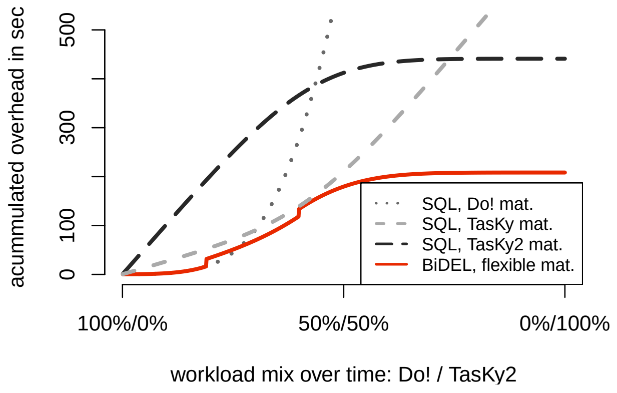
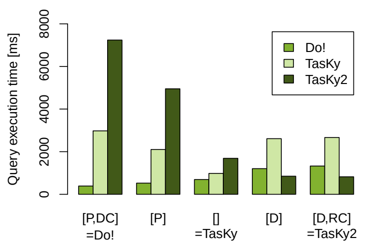


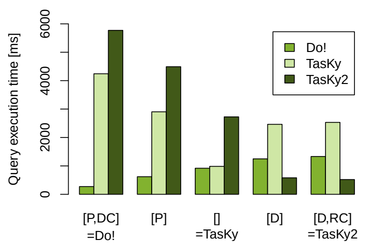
Fig. 13: Benefit of flexible materialization.

8.3 Benefit of Flexible Materialization

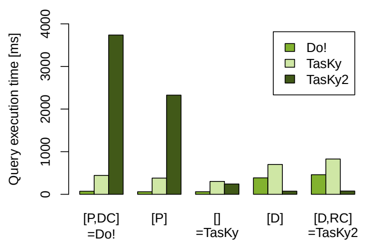
Although InVerDa introduces a slight overhead, the data independence provides large performance benefits. Adapting the physical materialization to the current



(a) Materializations for TasKy mix.



(b) Materializations for TasKy read.



(c) Materializations for TasKy write.

Fig. 14: Different workloads on all non-redundant materialization of TasKy.

workload is hard with handwritten SQL, but almost for free with InVerDa (1 LOC instead of 182 in our TasKy example). Let us assume a development team spares the effort for rewriting delta code and works with a fixed materialization. Then, InVerDa easily increases the overall performance, as we will show using the TasKy example and the Wikimedia scenario.

TasKy (macro benchmark): We use the TasKy example with 100 000 tasks. Figure 13 shows the accumulated propagation overhead for handwritten SQL with three different but fixed materializations (according to TasKy, TasKy2, and Do!) and for InVerDa with an adaptive materialization. Assume, over time the workload changes from 100 % accesses to Do! (and 0 % to TasKy2) until finally all users access exclusively TasKy2 following the Technology Adoption Life Cycle. We split the adoption into 1000 time slices; within each we run 1000 statements. The workload mixes 50 % reads, 20 % inserts, 20 % updates, and 10 % deletes. In such a scenario, InVerDa's flexible materialization naturally starts at Do!, but changes to TasKy after several users started using TasKy2, and finally to TasKy2 when the majority of users also did so. As soon as an evolved materialization serves the current workload better, we instruct InVerDa to change the materialization. As can be seen, InVerDa facilitates significantly better performance—including migration cost—than a fixed materialization.

TasKy (micro benchmark): The DBA can easily choose between multiple materializations—Figure 14 shows the data access performance on the three schema versions for each of the five non-redundant materializations of our TasKy example. The materialization schemas are represented as the lists of SMOs that are materialized, which we abbreviate: DC (`DROP COLUMN`), RC (`RENAME COLUMN`), D (`DECOMPOSE`), and P (`PARTITION`), so $[D, RC]$ corresponds to schema version TasKy2. 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 14(a), 100 % reads in Figure 14(b), and 100 % inserts in Figure 14(c) on the respective schema version. Again, the measurements show that accesses to each schema version are fastest when its table versions are materialized, i.e., when the physical materialization fits the accessed schema version. For instance, writing to TasKy2 is 49 times faster when the physical materialization matches TasKy2 instead of Do!. However, there are differences in the actual overhead, so the globally optimal materialization depends on the workload distribution among the schema version.

Wikimedia: We now measure the read performance in Wikimedia for the template queries from [9] 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 15, a large 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 data backwards.

8.4 Cost Model

We evaluate all evolutions with two SMOs as well as the TasKy example to analyze the learned cost model.

Evolutions with two SMOs: We consider all evolutions with two subsequent SMOs. Thereby, the first SMO always creates a table version $R(a, b, c)$, which is further evolved by the second SMO, so we end up

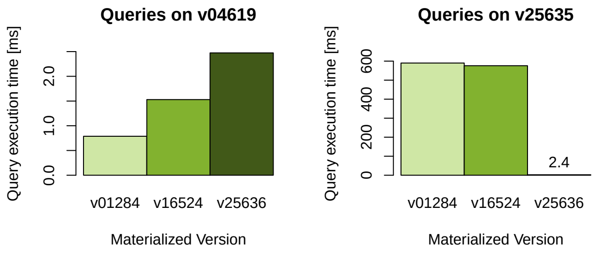


Fig. 15: Optimization potential for Wikimedia.

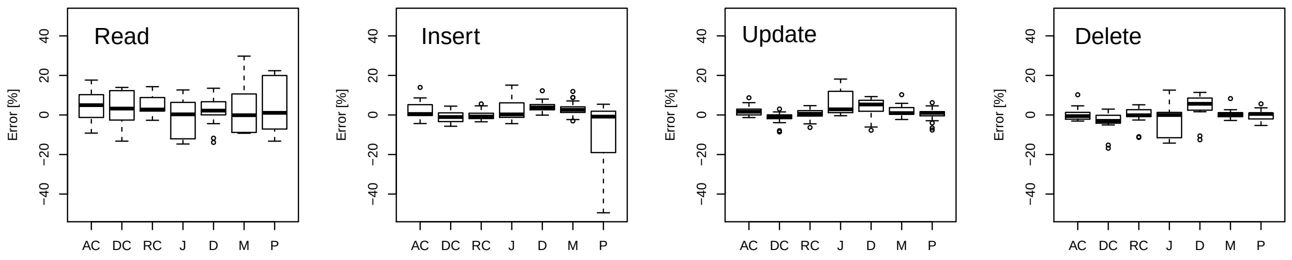


Fig. 16: Error of the cost estimation for two subsequent SMOs, with the second being an `ADD COLUMN` SMO.

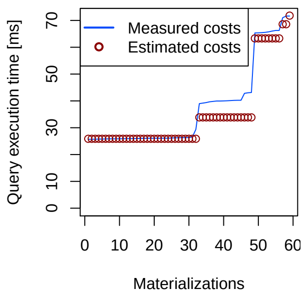
with three schema versions. We measure the costs for accessing the third schema version with the first being materialized and compare this to the estimated costs. Figure 16 shows the estimation error, which expresses the percentage of misestimation compared to the actually measured costs. We exemplarily show the estimation error for all scenarios that have an `ADD COLUMN` as second SMO; the first SMO is shown at the x-axis. The deviation 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. The deviation is close to zero for writing and roughly 10 % for reading, which is still reasonably precise. As comparison, typical estimation errors of common query optimizers are two to ten times higher and still allow finding good plans [27]. The estimation error for the remaining combinations is in the same range [13], which allows the advisor to safely rely on the cost model providing estimates in no time.

TasKy: The TasKy scenario allows 59 non-, partially, and fully redundant materializations. In Figure 17(a) we see the estimated and the measured costs for reading the Do! schema versions for all materializations, sorted by the measured costs. The cost model estimated the actual costs for reading the Do! schema version with a mean estimation error of 14 % and the order of the estimated costs is equal to the order of their actual costs, which allows the advisor to make very precise decisions. Figure 17(b) confirms these finding for a workload that mixes read and write operations on all three schema versions. The deviation is 24.2 % but the general order of the materializations is still estimated well. Thus, a materialization chosen with the help of the cost model, is very likely to perform well in practice.

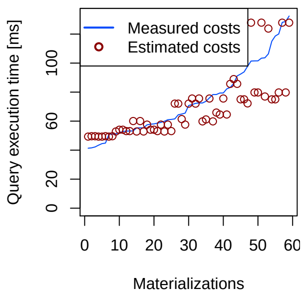
Since the optimization algorithm covers a very large search space, the time for calculating the costs of each specific materialization must be as short as possible. We measure the time required for the cost model-based estimation in comparison to physically migrating the database to a new materialization and measuring the actual costs for the given workload and the given materialization. We assume the TasKy version with 100 000 entries to be materialized. To get the costs with Do! being materialized, we need 8.21 s to change the materialization and another 33.32 s to measure the mean execution time for a read/write workload mix with 12 statements (read, insert, update, delete on all three schema versions respectively) each being executed five times on all schema versions. This sums up to 41.53 s for the whole example. In contrast, the cost model returns the estimated costs after 132.56 µs, which is a speedup of factor 2.5×10^5. This speedup further increases with growing table sizes, since the physical migration and analysis can easily last hours or days, while our cost model is independent from the table sizes. 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 was trained with 1000 to 100 000 tuples in 50 steps, which took less than ten minutes. In conclusion, the cost model proved to be sufficiently precise and fast to allow the advisor to make informed decisions.

8.5 Optimizer

The objective of the optimizer is to find the best possible materialization for a given workload on a given system. We run the advisor both in the TasKy scenario and in the Wikimedia scenario to evaluate its feasibility.



(a) Error of reading in Do!.



(b) Error of workload mix.

Fig. 17: Estimation errors in the TasKy example.

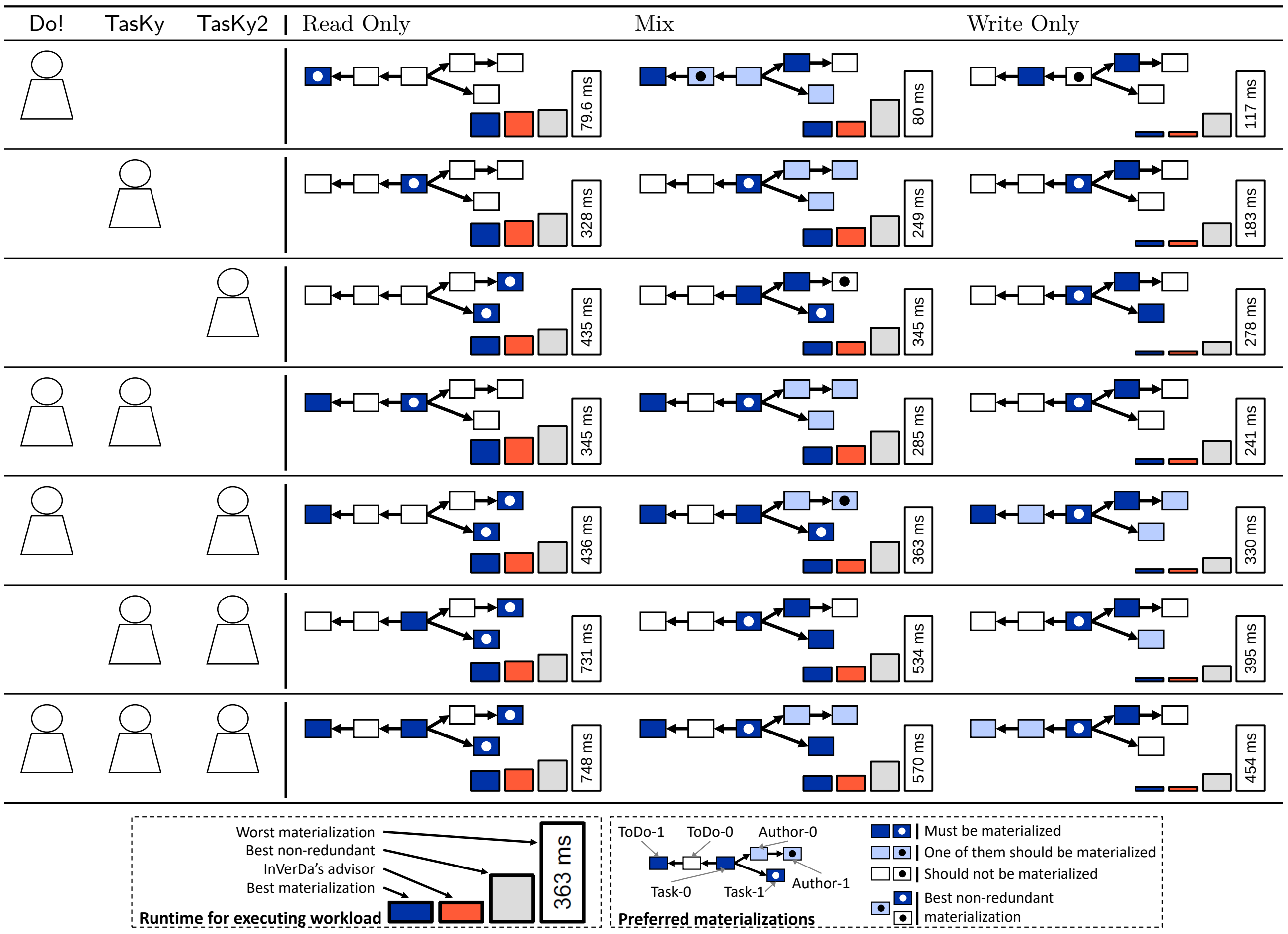


Fig. 18: advisor results for TasKy.

TasKy: The example is small enough to provide insights into the whole search space and a detailed analysis of the materializations. Figure 18 shows the results for running InVerDa's advisor for 21 different workloads structured in two dimensions: first, users access any non-empty subset of the schema versions (rows) and second, the workload comprises (1) only reading, (2) a 50:50 mix of reading and writing, or (3) only writing (columns). For each workload, we depict the best possible materialization (first bar) that should be found by the advisor. Since the advisor relies on the learned cost model, it might happen that the proposed materialization (second bar) may have slightly higher costs than the optimum. We further include the best non-redundant materialization (third bar) to analyze the benefits of partial redundancy as well as the worst possible materialization (fourth bar) for comparison. We deep dive into the best non- and partial redundant materialization and highlight the table versions that should be materialized, as explained in the key of Figure 18.

The measurements show that there is a **significant optimization potential**, since the worst materialization is always way more expensive than the optimum. For read operations the optimization potential is up to a factor of 5; for write operations the worst materialization can be up to 30 times more expensive than the best one. Fortunately, InVerDa's advisor always proposes a materialization that is almost as fast as the best materialization—it is never more than 10 % off, hence the **cost model works** for the given scenarios. The cost model-based optimization covers orders of magnitude more materializations in the same time, which compensates for the deviation in most scenarios.

Further, the best non-redundant materialization is up to a factor of 10 more expensive than the proposed partially redundant materializations. Contradicting a DBAs intuition, write operations can benefit from partial redundancy as well, since the number of auxiliary tables that need to be updated is reduced. This underlines that simply using the **naïve materialization** is often significantly **more expensive** than a partially redundant materialization proposed by the advisor.

Let us analyze the best materializations in detail starting with workloads on single schema versions (first three rows of Figure 18). When reading, we obviously materialize the accessed table versions to answer all queries locally. There is no need for redundancy since other table versions are not accessed either way. For writing, the best materializations are not that obvious, which supports the need for an automated advisor. The best non-redundant materialization always materializes 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—and 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. The best partially redundant materializations reduce the number of write operations on data and auxiliary tables even further.

For the workloads on multiple schema versions (last four rows of Figure 18), we see similar patterns. Read accesses perform best when all the accessed table versions are materialized. 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. Those non-redundant materializations that perform best for writing always materialize the initial `Task-0` table version, since 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 developer perspective. The proposed partially redundant materializations are significantly better than the naïve non-redundant materializations. This again emphasizes the need for an advisor. In summary, we confirmed that InVerDa's advisor 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.

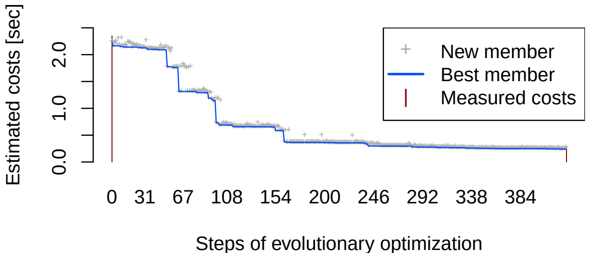


Fig. 19: Advisor for Wikimedia.

Wikimedia: We confirm our findings using the realistic scenario of Wikimedia. The 203 table versions allow a total of 1×10^{61} materializations. Figure 19 shows a run of InVerDa's advisor: After 828 ms with 429 evolved members in the population, there was no significant improvement over the last 30 steps, so the advisor 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. Further, it shows the measured costs for both the best initial materialization and the finally proposed one, which improved from 2.35 s to 0.26 s. The workload mixes read and write operations on schema version *v04619* (28th version) and *v25635* (171th version).

We do not know the global optimum, but we can safely say that InVerDa's advisor achieved an estimated speedup of a factor of 10 compared to the naïve initial materializations. We confirmed this speedup by physically applying the advisor's proposal. Besides several small improvements, there are few steps that significantly improved the overall performance by introducing redundancy to initially non-redundant members of the population. In other scenarios, it might be promising to search 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.

Summary: We have shown that BiDEL supports robust and agile database evolution and InVerDa generates delta code with a reasonable overhead. The guaranteed data independence facilitates performance improvements that can be easily exploited by the advisor. The cost model estimates the costs for a given workload very fast and feasibly well to allow the advisor informed decisions and to release DBAs from diving into the characteristics of data access propagation in MSVDBs.

9 Related Work

BiDEL and InVerDa are not the first attempts to support schema evolution. For practitioners, valuable tools such as Liquibase (www.liquibase.org), Rails Migrations (guides.rubyonrails.org/migrations.html), and DBmaestro Teamwork (www.dbmaestro.com) help to manage schema versions outside the DBMS and generate SQL scripts for migrating to a new version. They mitigate data migration costs, but focus on schema evolution with very limited support for co-existing schemas.

Both the database evolution [24,21] and co-existing schema versions [23] are well recognized in database research. Table 4 classifies related work and highlights the contributions of BiDEL and InVerDa. For database evolution, existing approaches increase comfort and efficiency, for instance by defining a schema evolution aware query language [22] or by providing a general framework to describe database evolution in the context of evolving applications [10]. Meta Model Management helps handling multiple schema versions “after the fact” by allowing to match, diff, and merge existing schemas to derive mappings between these schemas [5]. The derived mappings are expressed with relational algebra and can be used to rewrite old queries or to migrate data forwards—not backwards, though. In contrast, the inspiring works PRISM [8] and PRISM++ [7] propose to let the developer specify the evolution with SMOs “before the fact” to a derived new schema version. PRISM/PRISM++ are intuitive and declarative DELs that implicitly allow migrating data forwards and rewriting queries from old to new schema versions. As an extension, PRIMA [20] takes a first step towards co-existing schema versions by propagating write operations forwards and read operations backwards along the schema version history, but not vice versa. InVerDa also covers write operations on those former schema versions. BiDEL slightly extends the PRISM DEL to be relationally complete [14] and to be bidirectional at the same time [15]. According to our evaluation in Section 8.1, BiDEL SMOs are as compact as PRISM SMOs and orders of magnitude shorter than SQL.

These works provide a great basis for database evolution and InVerDa builds upon them to add **data independence** and **MSVDB support**, which basically requires bidirectional transformations [25]. Particularly, symmetric relational lenses facilitate read and write accesses along a bidirectional mapping [16], while auxiliary tables persist the complements to not lose any data [18]. For InVerDa, we adapt this idea to bidirectional SMOs. There are multiple systems also taking this step, however, the DELs are usually rather limited or work on different meta models like data warehouses [3]. The ScaDaVer system [26] allows additive and subtractive SMOs on the relational model, which simplifies bidirectionality and hence it is a great starting point towards more powerful DELs. BiDEL also covers restructuring SMOs and is based on established DELs [7,14]. To our best knowledge, BiDEL is the first DEL with SMOs intentionally designed and formally guaranteed to fulfill the symmetric lens conditions.

✓ supported ✗ not supported	SQL	Model Mngt	PRISM	PRIMA	ScaDaVer	Sym. Lenses	Mat. Views	**BiDEL**
Database evolution language	✗	✗	✓	✓	✓	✗	✗	✓
Relationally complete	✓	✓	✗	✗	✗	✗	✓	✓
Co-existing schema versions	✗	✗	✗	✓	✗	✓	✓	✓
- *Forward query rewriting*	✗	✓	✓	✓	✓	✓	✗	✓
- *Backward query rewriting*	✗	✗	✗	✗	✗	✓	✓	✓
- *Forward migration*	✗	✓	✓	✓	✓	✓	✓	✓
- *Backward migration*	✗	✗	✗	✗	✗	✓	✗	✓
Guaranteed data independence	✗	✗	✗	✗	✗	✓	✗	✓

Table 4: Contribution w.r.t. related work.

The optimization of the physical materialization is closely related to materialized view selection. Mistry et al. realize materialized view selection based on multi-query optimization techniques [19], and Agrawal et al. tuned a cost model-based approach to be robust and to scale [1], so it was incorporated into Microsoft's SQL server 2000 [2]. While InVerDa's advisor is inspired by those systems, its optimization problem has a different flavor: in MSVDBs there are no always-materialized base tables, but auxiliary tables depending on the chosen materialization that determine the performance. Further, there are efficient algorithms to update materialized views [12]—InVerDa applies similar techniques to propagate write operations to materialized table versions incrementally. In a word, there are many approaches tackling different aspects of MSVDBs as shown in Table 4. To our best knowledge, InVerDa's concepts are the first to facilitate MSVDBs end-to-end.

10 Conclusion

We started out with the goal to make database development just as agile as software development. Evolving a production database without compromising the data's correctness is a tough challenge in practice that accounts for significant costs in software projects. Often different stakeholders and different subsystems have different evolution speeds, which requires parts of a software system to exist in multiple versions at the same time. DBMSes do not provide co-existing schema versions these days, so developers end up writing delta code by hand, which is very expensive and error-prone.

To this end, we presented **InVerDa**, an **MSVDB** that facilitates co-existing schema versions in a single database. New schema versions can be easily created with our **Bidirectional Database Evolution Language BiDEL** that couples the evolution of both the schema and the data in intuitive and compact SMOs. BiDEL is **relationally complete**, so there is no need to fall back on SQL. Further, BiDEL is **bidirectional** with a materialization-independent mapping semantics, which allows us to generate delta code in an MSVDB automatically—according to our evaluation, developers write orders of magnitude less code with BiDEL compared to traditional SQL. MSVDBs provide **full data independence**, hence the DBA can freely move or replicate the materialization along the schema versions history. We formally guarantee that each single schema version behaves like a regular database irrespective of the chosen materialization.

This opens up a huge space of possible materializations that can speed up read and write operations by orders of magnitude. Therefore, we equip DBAs with a **cost model-based advisor** to optimize the materialization for the current workload. The cost model is individually learned and proved to be precise enough to gain significant performance improvements.

With MSVDBs, we speed up database development to the pace of modern agile software development. Plus, there are even more scenarios where the presented concepts come in handy, e.g., 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 [13]. Further, there are promising future research questions such as zero-downtime migration, extending BiDEL's expressiveness, or speeding up data access propagation by fine-tuning the management of auxiliary information and by combining and reducing the data access propagation through chains of SMOs.

Acknowledgements This work is partly funded by the German Research Foundation (DFG) within the RoSI RTG.

References

1. Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.
2. 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.
3. Meenakshi Arora and Anjana Gosain. Schema Evolution for Data Warehouse: A Survey. *IJCA*, 22(5):6–14, 2011.
4. Peter Bentley and David Corne. *Creative evolutionary systems.* Morgan Kaufmann, 2002.
5. Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, 2007.
6. Michael L. Brodie and Jason T. Liu. Keynote: The Power and Limits of Relational Technology In the Age of Information Ecosystems. In *OTM*, 2010.
7. Carlo Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *VLDB Journal*, 22(1):73–98, 2013.
8. Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *PVLDB*, 1(1):761–772, 2008.
9. Carlo Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *ICEIS*, 2008.
10. Eladio Domínguez, Jorge Lloret, Ángel L. Rubio, and María A. Zapata. MeDEA: A database evolution architecture with traceability. *DKE*, 65(3):419–441, 2008.
11. 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.
12. Ashish Gupta and Inderpal Singh. Mumick. *Materialized views: techniques, implementations, and applications.* MIT Press, 1999.
13. Kai Herrmann. *Multi-Schema-Version Data Management.* Ph.d. thesis, http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-231946, Technische Universität Dresden, 2017.
14. Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. CoDEL – A Relationally Complete Language for Database Evolution. In *ADBIS*, 2015.
15. 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 *SIGMOD*, 2017.
16. Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, 2011.
17. Philip Howard. Data Migration Report, Bloor, 2011.
18. James McKinna. Complements Witness Consistency. In *Bx*, 2016.
19. Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, 2001.
20. Hyun J. Moon, Carlo Curino, Myungwon Ham, and Carlo Zaniolo. PRIMA – archiving and querying historical data with evolving schemas. In *SIGMOD*, 2009.
21. Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *ACM SIGMOD Record*, 35(4):30–31, 2006.
22. John F. Roddick. SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(2):10–16, 1992.
23. John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.
24. Ioannis Skoulis, Panos Vassiliadis, and Apostolos Zarras. Open-source databases: Within, outside, or beyond Lehman's laws of software evolution? In *CAiSE*, 2014.
25. James F. Terwilliger, Anthony Cleve, and Carlo Curino. How Clean Is Your Sandbox? In *ICMT*, 2012.
26. Bob Wall and Rafal Angryk. Minimal data sets vs. synchronized data copies in a schema and data versioning system. In *PIKM*, 2011.
27. Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 6(10):925–936, 2013.