Versioning in Main-Memory Database Systems
From MusaeusDB to TardisDB

Maximilian E. Schüle
maximilian.schuele@tum.de

Lukas Karnowski
lukas.karnowski@tum.de

Josef Schmeißer
josef.schmeisser@tum.de

Benedikt Kleiner
benedikt.kleiner@tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT
As relational database systems do not support collaborative dataset editing, online lexicons—such as Wikipedia’s MediaWiki—build their own version control above the database system to allow constraint-preserving version checkouts or commits involving multiple tables. To eliminate the need for purpose-specific solutions, we propose adding version control as a layer on top of the database system or integrating versioning in the database system’s core.

This paper presents the first two architectures for versioning an entire state of a database system with respect to references among multiple relations. We design the prototype MusaeusDB as a solution for existing database systems, either as an external tool or as an extended SQL interface. The prototype TardisDB—an extended main-memory database system—reuses multi-version concurrency control for in-place updates while keeping older versions accessible. For performance tests on different storage layouts, we create—based on Wikipedia’s page history—the TardisBenchmark. Our results show that it is indeed feasible to reduce wasted space while still ensuring constant retrieval time. Also, extending a main-memory database system’s multi-version concurrency control has no negative impact on the transactional throughput. For further research on database versioning, we offer a flexibly sized benchmark with time evolving, text-based datasets and compression techniques.

CCS CONCEPTS
• Information systems → Database management system engines; Main memory engines
• Applied computing → Version control

KEYWORDS
Version control, SQL

ACM Reference Format:

Figure 1: Sketch of the trade-off between storage savings (compression rate) and retrieval time: storing only one version snapshot and computing the others out of the changed differences (diffs) will reduce the amount of storage needed but will increase the retrieval time.

1 INTRODUCTION
On the one hand, software engineering relies on version control systems such as SCCS, CVS, SVN and Git to allow distributed code development and to document the project’s progress in commits and version tags. On the other hand, database systems are essential for efficiently handling huge amounts of data in index structures such as B*- [2] or Adaptive-Radix-Trees [13]. To unify both use cases, approaches such as temporal databases or dataset versioning aim to bind the validity of tuples to time instances or to allow collaborative dataset editing.

The most common example for documenting versions inside a database system is MediaWiki—the wiki software behind Wikipedia—as it manages the versions in a page content table of a MariaDB system for every article. So far, research has focussed on versioning single datasets. But to fulfill the requirements for versioning a wiki system, a version control for database systems should be able to include multiple relations per single commit and respect referential integrity. When versioning one namespace of the Wikipedia encyclopedia, we expect a compression rate of up to 70 % when storing the differences only. Our estimation is based on the Simple English Wikipedia edition for which we computed the file and the edit differences for all page revisions (see Table 1). Applying versioning and compression techniques to database systems eliminates...
we present solutions for overcoming the trade-off between storage savings and retrieval costs.

The paper is structured as follows: after a review of related work the need for purpose-specific solutions, reduces the storage needed and allows comparable retrieval times (see Figure 1). We claim that database systems with full incorporated version control commands fill the gap caused by distributed data and knowledge management, and facilitate the versioning of wiki entries.

Starting at extending single tables—known from OrpheusDB—to multiple relations per version, we present MusaeusDB as a stand-alone tool along the database system. It implicitly sends SQL commands, manages the versions in meta tables and benefits from the database systems’ user rights management, as tables are checked out in the user’s namespace. Based on the same schema, MusaeusSQL incorporates the SQL commands by providing a single interface for SQL as well as for versioning commands. We make use of the gathered knowledge for TardisDB, a main-memory database system that deeply integrates versioning based on additional bitmaps for each tuple and reusing multi-version concurrency control to preserve former databases’ states. In our TardisBenchmark, we evaluate TardisDB and the different storage approaches developed for database systems, and compare their runtime to those of the version control system Git. The main contributions of this work are:

- Extending version control on top of database systems to include multiple tables instead of single datasets per version
- The architectural blueprint for integrating versioning inside a main-memory database system
- A benchmark, which covers dependencies between relations and evaluates different storage techniques in respect for the querying performance

The paper is structured as follows: after a review of related work on temporal databases and their connection to research on version control, we will introduce our SQL-based prototypes (MusaeusDB, MusaeusSQL) and their architecture comprising the schema, syntax for the user input and storage conventions. We proceed with the architecture of TardisDB, for which we adapt the table scan operator and the multi-version concurrency control of a main-memory database system to allow branches and to store multiple states of the database. To benchmark the prototypes, we will explain how we extract Wikipedia’s page edit history dump of October 1, 2018.

Table 1: Estimation of saved storage when using compression techniques based on the Simple English Wikipedia page edit history dump of October 1, 2018.

<table>
<thead>
<tr>
<th>Compression Technique</th>
<th>Size</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Page Edit History</td>
<td>35.0 GiB</td>
<td>-</td>
</tr>
<tr>
<td>Current Version Only</td>
<td>1.1 GiB</td>
<td>-</td>
</tr>
<tr>
<td>History as File Diffs</td>
<td>14.0 GiB</td>
<td>59.77 %</td>
</tr>
<tr>
<td>History as Edit Diffs</td>
<td>9.4 GiB</td>
<td>72.71 %</td>
</tr>
</tbody>
</table>

2 RELATED WORK

Whereas version control for software projects has a long tradition, studies on database systems have mainly focussed on temporal databases. This section describes temporal databases and research on full versioning of databases (mostly limited to single datasets), the starting point of this work, and Git as the most popular version control system, which is later used as competitor.

2.1 Temporal Databases

Temporal databases bind the validity of tuples to time intervals by offering an additional datatype. Until 2011, several extensions of SQL such as TQuery [20] and SQL2 [21] examined adding time travelling to fetch the database state from a certain date in the past. They tried to be backward compatible with SQL:92 but were not considered for a new SQL standard. Instead, the SQL:2011 standard [12] marks the tuples of a temporal database with two date columns indicating the period of the tuple’s validity. Nowadays, most of the commercial database systems support time travelling. For example, Microsoft SQL Server (MSSQL)¹, IBM DB2 10² and MariaDB³ take a time range from two date columns. PostgreSQL (PSQL)⁴ offers the tstzrange type to indicate time instances and an additional history table for past records. Based on the presence of data types for temporal databases, different optimising techniques aim at integrating time travelling inside transaction handling [14] or at indexing tuples for accessing time evolving data efficiently [18]. Beyond relational database systems, studies focus on compressing time evolving data as XML archives [7] or on integrating time travelling inside RDF databases to be addressed with SPARQL [23].

2.2 Version Control: Git

Git⁵ was initially developed for managing large software projects, but can be used for other purposes as collaborative text editing. Internally, commits represent changes; one commit may have multiple preceding ones, forming an acyclic graph. It works decentrally and allows branching and merging for collaborative code development.

Actually, Git is a key-value store for storing any kind of data returning a unique hash identifier, which allows the data to be retrieved. Internally, Git stores three type of objects, blob, tree and commit. A blob object represents the content (binary large object) that Git has to track, so every object corresponds to one file. Consequently, for every minor edit, a new blob object is created. The tree objects correspond to the directories and store the identifiers of the contained blob or sub-tree objects. Finally, the commit objects contain a parent one and the tree objects.

This so-called Loose Object Format would be inefficient if no packing took place. But Git uses delta encoding, a form of taking differences between objects. The optimal differences, adjustable by one variable that sets the maximum length of a delta-chain and one for the number of files to compare, are stored in packfiles. This information is useful for adapting delta encoding and for understanding the benchmark results.

¹ www.mssqltips.com/sqlservertip/5680/introduction-to-sql-server-2016-temporal-tables/
² www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/
⁴ https://wiki.postgresql.org/wiki/SQLO1Temporal
⁵ https://git-scm.com/
2.3 Versioning in DBMS

In contrast to temporal databases, which protocol the validity of each tuple, versioning should log at a high granularity to preserve the whole database's state. As part of the DataHub system [3, 4], Decibel [15] is an approach of integrating dataset versioning inside a database system. It benefits from recovery, "fault tolerance" and SQL as the declarative language provided by the database system, but also includes VQuel [8] as a versioning query language besides SQL. It evaluates different bitmap based storage techniques for creating and merging branches by using a self-created versioning benchmark. The versioning scope is limited to single relations. We take the idea of bitmaps indicating included tuples for each branch of a main-memory database system. To test our system, we adapt the versioning benchmark, as well.

A version control on top of database systems is OrpheusDB [10, 24], called a "bolt-on" technique as it works on existing datasets. First, the data to be versioned is loaded from CSV files or from a database system into an extended database schema. Then, every tuple is extended by a record identifier rid. A version consists of multiple rids, which are managed in a separate table. The database system itself remains unmodified as OrpheusDB can run on top of any arbitrary database system as long as it provides SQL-92 commands and an array datatype. Our SQL prototype, MusaeusDB\(^6\), will extend this work to manage multiple tables. Independent of the underlying data storage, RStore [6] allows versioning on top of arbitrary key-value stores. It investigates the trade-off between storage costs, query performance and online updates. It uses delta compression—as in this work—to compress textual documents.

A key challenge for versioning datasets—postulated in 2015 [3]—is the trade-off between reducing the amount of storage with delta compression while restoring datasets fast enough. To tackle the trade-off, array database systems, designed to host array-like datasets, incorporate versioning techniques as forward or backward delta compression [22]. SciDB [19] decides on a minimum weight spanning tree either to materialise versions for fast recreation or to store the differences for dense as well as sparse array-oriented data. We will use delta compression for text-like data.

\(^6\)Musaeus is a contemporary of Orpheus; in honour of OrpheusDB.

\(^7\)https://github.com/aergoio/litetree

3 MUSAEUSDB: VERSIONING USING SQL

The SQL prototypes—called MusaeusDB and MusaeusSQL—rely on the extended schema of OrpheusDB to combine changes over multiple relations as one version. The key idea is to keep the initial data tables unchanged but with an added record identifier (rid) and to store the information about versions in separate tables. As multiple tuples across different versions may share the same primary key, the rid is needed as the new primary key for all tuples.

The version table manages the corresponding rids for each version and each table (we adopt the rlist as array-like structure but the table can be easily normalised using unnest). For each commit, the meta table contains information such as the commit message or the parent version. In comparison to OrpheusDB, we extended the version table by one attribute referencing the table in question. Instead of only one tuple, it contains as many tuples per version as tables involved. During query processing, we need an additional join predicate to retrieve the tuples of a certain version. Figure 2 shows the tables responsible for managing the commits and the tables containing the data. In this example, the meta table hosts two commits, one initial and one descending; the tuples concerned are stored in the version table. For example, the name of the users has changed after they got married, so the user table’s attribute name has been updated and “wedding” was added to the task table. The data tables now have a unique artificial key (vid) as the original keys will not suffice as primary keys, but are restored on a table checkout. Indexing rid as primary key allows fast join processing even for a huge number of entries. Given the presented database schema, we present MusaeusDB as a tool besides and MusaeusSQL as a tool on top of an existing database system.

3.1 MusaeusDB as a Separate Tool

MusaeusDB stores datasets in “public repositories” and allows users to clone them. It benefits from the conception of database systems as they provide multiple databases per server (at least one for each user) with named schemas per database. In fact, a “public repository” is an own namespace (a named schema) of a public database. Each user is allowed to check out a certain version in a private namespace of his/her database and may modify the tuples locally. Afterwards, the changes get propagated by a commit to the origin repository.

In Figure 3, we see the distinction between public and user databases with separate namespaces for the repositories. Each namespace represents one repository. The default namespace of the public database hosts all versions and their meta information. On a checkout, all tuples belonging to the dedicated version are
checked out into tables that are created for that purpose in the specified namespace. To keep track of all tables currently checked out, all checkouts are documented in the default namespace of the user’s database. MusaeusDB is based on database systems implementing the PostgreSQL interface and works as a separate tool. For commits and checkouts it uses the pqxx library\(^8\) to connect to the database server. The source code has been made publicly available\(^9\). In the following, we will introduce the SQL commands behind init, checkout and commit.

### 3.1 init

The init command prepares the tables of an existing namespace for versioning comparable to git init. The command expects the name of the destined global repository and the name of a schema with its tables to be prepared for versioning and collaborative working (see Listing 1). MusaeusDB generates the SQL commands to endue each tuple with an rids as its new primary key and to create a global table in the designated global namespace to which each of the source’s relations can be copied.

#### 3.1.1 init

The init command prepares the tables of an existing namespace for versioning comparable to git init. The command expects the name of the destined global repository and the name of a schema with its tables to be prepared for versioning and collaborative working (see Listing 1). MusaeusDB generates the SQL commands to endue each tuple with an rids as its new primary key and to create a global table in the designated global namespace to which each of the source’s relations can be copied.

#### 3.1.2 checkout

The checkout command works contrary to the init one: this takes the name of the global namespace and copies the tables to newly created ones in a designated private namespace (see Listing 2). The global rids is hereby omitted and the original primary keys are restored.

#### 3.1.3 commit

The commit command updates the global repository with changed, inserted or deleted tuples. It takes the name of the source’s namespace and a commit message (see Listing 3). It treats all changes as one whole commit and pushes the updates to the origin. The rids for the new version is copied from the ancestor one except the rids of the changed or deleted tuples. For every changed or inserted tuple, a new rids is created and added to the rlist. As the command is translated into one atomic transaction, the tool ensures that commits are processed atomically, and respects referential integrity as local tables inherit the parent’s database schema. This is useful as MediaWiki stores the page title separately from the content, but using references.

### 3.2 MusaeusSQL: Using One Interface

MusaeusSQL is a lightweight tool on top of existing database systems and provides an interface for SQL as well as versioning commands. With MusaeusSQL, we tackle the obstacles arising by using a separate tool: two different user interfaces and the doubled configuration setup. Instead of checking out the relations locally before any edit, MusaeusSQL performs actions directly on the remote repository.

#### 3.2.1 Design

The conceptual design stays the same: on a commit, an entry consisting of a new vid and the corresponding rids is added to the version table. But now, the rids are created as soon as a tuple is inserted or created. As the rids are hidden from the user perspective, MusaeusSQL translates SQL queries to restrict the validity of a query to the tuples of the current version.

MusaeusSQL makes it easy to create extensions to support further database systems or SQL commands that are currently not supported (see Figure 4). Part of its architecture is made up of SQL/versioning operations and a class for the communication with the database server. Our SQL transformations are based on objects

---

\(^8\) http://pqxx.org/development/libpqxx/

\(^9\) https://gitlab.db.in.tum.de/tardisDB/musaeusDB

---

Figure 3: Distinction between global and local (user) space in MusaeusDB: The global space maintains a separate namespace for each repository, relations can be checked out for modifications in the user’s namespace.

Figure 4: Architecture of MusaeusSQL: Operations are divided into basic SQL and versioning commands; SQL commands are transformed as the extended schema is hidden, versioning commands are translated into SQL queries.
of the Hyrise [9] C++ SQL Parser\(^{10}\) and are redirected as strings to the database connector. We will describe the transformation of select, insert, update and delete statements in the following.

3.2.2 Query Transformations. In Listing 4, we see the transformation of a select statement: first, we pick the id of the version currently checked out. Then, we transform each table into a subquery containing only the relevant tuples. Thus, we perform a join on the rid attributes of the version table.

The remaining statements were transformed in a similar manner: an insert statement adds an rid to the next version (see Listing 5). Instead of removing tuples, a delete statement just removes the corresponding rid of the new version (see Listing 6); update is transformed to an insert query to preserve the former state.

3.2.3 Versioning Commands. In addition to common SQL-92 queries, MuisaesSQL offers the commands commit <message> and checkout <version> also known from Git. The checkout command updates the local version to be used. All subsequent SQL commands will use this state when reading or modifying data. The commit command materialises a state. Before committing, all tuples belong to a temporary state. After committing, the state will be persisted and a new temporary state created.

4 TARDISDB: VERSIONING INSIDE A MAIN-MEMORY DATABASE SYSTEM

Having summarised how versioning can be performed on top of database systems, this section describes how versioning can be integrated inside a modern main-memory database system. Therefore, we cover how version control is realised inside a prototyping framework of a code-generating in-memory database system. This framework utilises the push operator model [16] and represents the logical core of a main-memory database system. Query plans are compiled to LLVM’s Intermediate Representation (IR), then optimised and executed.

The push model, as the name suggests, is characterised by the inversion of the logical tuple flow. Tuples are pushed from a child operator to its parent rather than pulled as in the traditional approach. This model can generally be characterised by the two functions produce() and consume(attributes, source). A parent operator will request tuples from its child by invoking produce. In response to the produce call, the child operator will generate its own tuples. These tuples are subsequently pushed by invoking the consume function of the parent operator. This leads to an important difference in contrast to the traditional pull model: in the push model, a child always passes all of its tuples to its parent at once, rather than a single tuple at a time.

For TardisDB\(^{11}\), we modify the table scan operator to push only visible tuples. We therefore introduce branches in the form of bitmaps with every set bit representing an active tuple in the certain branch, as well as we adapt multi-version concurrency control (MVCC) to retrieve any previous state of a tuple.

4.1 Bitmaps for Versioning

TardisDB is based on the tuple-first approach described by Madrox et. al. [15], where all tuples are stored in one table and bitmaps indicate the association to a certain branch or version. Each branch consists of only one version, so branching and versioning (updates, insertions, deletions) means the same operation, that is, copying the bitmap to serve as the new starting point. On an insert, the bits corresponding to the inserted tuples will be set; on a delete, the corresponding ones unset and both an update.

---

\(^{10}\)https://github.com/hyrise/sql-parser

\(^{11}\)Time and Relative Dimensions in Databases: versioning (time) and branching
parent operator, as well as corresponding instructions for extracting the branch indicator bit from our branch bitmap. Listing 7 depicts the modified code generation logic within the table scan operator. The concept of bitmaps is easily transferable to multiple relations, we just have to maintain one bitmap for every branch, for each relation.

Nevertheless, we need to intersect the involved bitmaps to obtain a version chain in this approach, as no information is stored than the tuple itself. Also, many updates will result in sparse bitmaps. Thus, we will use bitmaps for branches only and will rely on multi-version concurrency control to track versions.

### 4.2 Reusing MVCC for Versioning

As database systems come along with concurrency control mechanisms to encapsulate transactions and to restore previous states, we can rely on these mechanisms to preserve different versions. We base our versioning approach on the multi-version concurrency control model [17] where updates happen in place and previous versions are stored in undo buffers. We therefore introduce the concept of a prioritised branch—so called master—with in-place changes (see Figure 6). We should thus be able to retain the high scan performance of our system in cases where only a few tuples are not active in the master branch.

Each branch is represented by a bitmap that indicates the active tuples. Creating a branch preserves the current version and simplifies its retrieval. Previous versions to which other branches still refer are chained in buffers. Similarly to [17], we maintain timestamps $ts(t)$ for every tuple, as well as for every branch, to indicate their creation. To map timestamps and tuples to a certain branch, we introduce markers for every tuple to indicate the creator branch ($created()$). This allows us to traverse the visible range of the tuple’s history. On creation of a branch, we copy the bitmap for the descending branch; all its tuples are visible. Afterwards, inserts or updates on the ancestor branch are hidden for the new one. To access the latest version of a tuple in a given branch, we first check the bitmap to ascertain whether a tuple is included, then we follow the chain until we reach an entry for a tuple that was created by the current branch or by a parent one. Formally, for each entry $t$ and each branch $b$ we can define the predicate $active(t,b)$ that evaluates to true when an entry belongs to the current branch:

$$active(t,b) \equiv created(b,t) \lor \bigvee_{p \in \text{parent}(b)} active(t,p) \land ts(t) < ts(b_p).$$

An entry is active within a branch when it was created by the respective one itself or by one of its parents ($\text{parent}(b)$). Of course, an entry, created by an ancestor branch, is only visible when it was created before branching. That is why we compare the entry’s timestamp with the branch’s one. When the chain is traversed, the first—newest—active entry will be returned.

The actual implementation considers a non-recursive reformulation of $active()$. We introduce the set-oriented mapping $C(b)$ of a branch $b$ that returns the lineage as pairs ($\langle \text{parent}, \text{child} \rangle$):

$$C(b) = \bigcup_{p \in \text{parent}(b)} \{ (p, b) \} \cup C(p).$$

The implementation uses the expression for creating a precomputed hashable out of the non-recursive definition of $active()$:

$$active(t,b_q) \equiv created(t,b_q) \lor$$

$$\exists(p,b_p) \in C(b_q) : created(t,p) \land ts(t) < ts(b_p).$$

Therefore, we use $C(b)$ to identify the relevant branching point that has to be investigated. If $ts(t) < ts(b_q)$ holds, we can infer that all timestamps of further descendants are also greater.
5 TARDISBENCHMARK

This section presents the TardisBenchmark based on the Wikipedia page edit history. We have chosen the MediaWiki schema as it is the most common example of a versioned database with reference key constraints, which we will introduce in the following. Afterwards, we will explain the different aspects and operations that we consider for our benchmark and what storage approaches we tested with the operations. We have published the benchmark as open-source\[12\].

5.1 MediaWiki Schema and Wikipedia Data

Wikipedia is the 5th most important web site in the world with 1,333,742 sites linking in [1]. Being a collaborative online encyclopedia to which everyone is allowed to contribute, Wikipedia relies on version control mechanisms that allow previous versions to be restored if the free edit rights are misused. Together with the fact that data and the architecture are freely available, Wikipedia is the ideal candidate for deriving a benchmark out of it.

The core of MediaWiki, the software behind Wikipedia, is a database schema (see Listing 8) out of the three relations page (the meta data of the articles), pagecontent (the actual content) and revision (references to former versions of an article). A current article can be retrieved by finding the page by title first (see Listing 9), then selecting the minimal timestamp of a revision in the table of revision. On a page edit, we just update the references.

A current article can be retrieved by finding the page by title first (see Listing 9), then selecting the minimal timestamp of a revision in the table of revision. On a page edit, we just update the references.

The new content is always (on update or insert) added to pagecontent. When the page is initially created, a new tuple is added in revision and page. Otherwise on a page edit, we just update the references.

5.2 Benchmark Operations

Our benchmark covers the most common use cases, which are the insertion of articles or retrieval of the latest or first version of an article. Figure 7 shows the conception of the benchmark with the XML file as input and the three different queries. The motivation for picking this set of benchmark operations is as follows:

- Insert. When a page is initially created or updated, the page content will be inserted as a new tuple to the database system. So inserting articles is the fundamental operation on which any project relies on. We take the Wikipedia dumps with full page edit history as real world examples.

- Retrieve Latest (retLatest). The common case an online encyclopedia is used or collaborative work is performed is the retrieval of the latest version of an article or the software. When using Wikipedia, users are always redirected to the current version. Former versions can be viewed on demand afterwards. So retrieving the latest version will form the majority of the work load.

- Retrieve First (retFirst). The superficial use case of retrieving the first (oldest) version of an article is the most challenging one. It covers retrieval of any prior version. This is important when storing former versions as differences to the latest article, as all deltas between the versions have to be applied. As another benefit, when we support the operations for retrieving both the first and the latest version of an article, we obtain the runtime of backward and forward delta-based versioning techniques, but we need to compute the differences in one direction only.

5.3 Storage Approaches

We consider the following four storage approaches:

5.3.1 Snapshot. The snapshot-based approach forms the classical way of storing every article as a new tuple in the manner of MediaWiki. We therefore rebuild the database schema out of pagecontent, revision and page to benchmark the performance. The operations are listed in further detail here:

- Insert. The new content is always (on update or insert) added to pagecontent. When the page is initially created, a new tuple is added in revision and page. Otherwise on a page edit, we just update the references.

- Retrieve Latest. Retrieving the latest version means a join on page (stores the latest pageid) and pagecontent.

- Retrieve First. We retrieve the first version of an article by selecting the minimal timestamp of a revision in the table of that name.

5.3.2 Diff. The difference-based versioning reduces the amount of wasted storage by storing the latest version as a whole and the previous ones as text differences (see Listing 11). We adapt the MediaWiki schema to make it suitable for storing text differences, as we save the latest page content as an additional attribute of the page table and store the text differences in pagecontent. As the latest version is stored as a whole in page, this ensures that the page demanded most frequently can be retrieved quickly, with no need for joins. For an arbitrary version, we apply all differences starting from the latest, also called backward deltas.

- Insert. On a page edit, the content in page will be updated and the difference from the previous one will be inserted in pagecontent. Initially, when an article is created, an empty tuple is inserted in page.

---

\[12\]https://gitlab.db.in.tum.de/tardisDB/TardisBenchmark
The evaluation section discusses the benefits and obstacles arising from the different storage approaches with respect to the different benchmark operations. Therefore, we evaluate the results of MusaeusDB and the TardisBenchmark—both text-based benchmarks—and conclude by finding the approach that is the most suited to certain use cases. In addition, we reproduce the branching benchmark of [15], for which code snippets have been made publicly available, to show comparable results of our bitmap-based versioning of TardisDB.

6 EVALUATION

The evaluation section discusses the benefits and obstacles arising from the different storage approaches with respect to the different benchmark operations. Therefore, we evaluate the results of MusaeusDB and the TardisBenchmark—both text-based benchmarks—and conclude by finding the approach that is the most suited to certain use cases. In addition, we reproduce the branching benchmark of [15], for which code snippets have been made publicly available, to show comparable results of our bitmap-based versioning of TardisDB.

6.1 MusaeusDB

To evaluate MusaeusDB, we have chosen two settings: In order to measure the impact of extending versioning to support multiple tables, we have reimplemented versioning on single datasets as described for OrpheusDB, but using C++ instead of Python to be compiled instead of interpreted. We omit MusaeusSQL as it is based on the same database schema and performs similar query transformations. 10^6 tuples of fictional users and corresponding tasks served as test data. We first initialised the schema by adding the rid and a version table, then we performed a checkout, updated every tuple and committed the changes. For the second setup, we compared the speed up gained from the main-memory database system HyPer [11] in contrast to PostgreSQL as the underlying database system. We performed the test on a Debian 9 machine with four cores of Intel i7-7700HQ CPU, running at 2.80 GHz clock frequency each, and 16 GiB RAM. As we see in Figure 9, MusaeusDB needed twice as much the time as OrpheusDB, as it has to keep track of two tables for each version. When HyPer was used as the underlying database system instead of PostgreSQL, we only needed less than half the amount of time, in average.

6.2 Branching Benchmark

In the style of the versioning benchmark [15], our branching benchmarks aims to evaluate TardisDB with respect to the scan performance in the presence of multiple branches. The benchmark itself is divided into a fixed number of epochs, each defined and concluded by the creation of a new branch (see Figure 8a). In total, 5 * 10^5 new tuples will be inserted during each epoch. Additionally, existing tuples will be updated, yielding an update-to-insert ratio of 15 to 1. The final branch hierarchy with the creation timestamp and the number of visible tuples per branch is depicted in Figure 8b.

During each epoch, our master branch takes a privileged role in that each time an operation is performed, a random branch is drawn from a distribution where Pr(branch=master) = 0.7 holds. This choice should, to a certain degree, resemble real-world scenarios in which operations on the master branch are more frequent. The probability of encountering a tuple belonging to the master branch during the scan benchmark is thus higher than for other branches. We used an Ubuntu 18.04 LTS server with an Intel Xeon CPU E5-2660 v2 processor with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM.

Our benchmark results are depicted in Figure 8c. As expected, scanning the master branch yields the best performance. This is particularly the case for the \texttt{retLatest} scan where no version chain...

---

**Listing 8:** Simplified MediaWiki schema: `page` (all meta informations), `pagecontent` (actual page content) and `revision` (history of former versions).

**Listing 9:** Retrieve page by given title ($\$1\$$).

**Listing 10:** Retrieve page content by identifier $\$1\$$.

**Listing 11:** Modified schema (Diff): `page` contains the current content, ` revision` manages the differences stored in `pagecontent`.  

- **Retrieve Latest.** Retrieving the latest article works in less time as no join must be performed anymore. The content attribute of `page` can be read directly.
- **Retrieve First.** The downside when retrieving the first article is that it has to be computed out of all differences in `pagecontent`, starting from the latest version of `page`. This operation has linear complexity with the number of edits.

5.3.3 Git. As we use the version control software Git as our competitor, we store all pages on the file system with one file per page and one commit per version. Traversing the predecessors of a commit works efficiently in Git. We store every article on a separate branch to avoid long chains when retrieving an earlier version.

- **Insert.** For every new page encountered, we create a new branch with the same name as the article descending from an empty commit (stub). Every commit of a branch represents a page edit.
- **Retrieve Latest.** Retrieving the latest version is identical to a look up to the latest commit of the respective branch.
- **Retrieve First.** We retrieve the first version by traversing all commits to the origin. The time complexity is linear with the number of commits.

5.3.4 LiteTree. LiteTree allows branches to be created for each new version of a page. It stores every version uncompressed, which allows any arbitrary article to be retrieved efficiently. We rely on the internal versioning mechanisms, so we just need one `page` relation for the content.

- **Insert.** A page edit is translated to an SQL update statement and a page creation to an SQL insert statement.
- **Retrieve Latest/First.** As the commits are enumerated for every branch, we can access and retrieve any commit.

---

**CREATE TABLE page (page_id INT PRIMARY KEY, page_title TEXT, page_latest INT);**

**CREATE TABLE revision (rev_id INT PRIMARY KEY, rev_page INTEGER REFERENCES page (page_id), rev_text_id INTEGER, rev_parent_id INTEGER, rev_timestamp TIMESTAMP);**

**CREATE TABLE pagecontent (old_id INT PRIMARY KEY, old_title TEXT, old_text TEXT);**

**Listing 10: Retrieve page content by identifier $\$1\$$.**

**Listing 9: Retrieve page by given title ($\$1\$$).**
We used 2 TB of SSD storage to get the best performance out of the CPU E5-2660 v2 with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM. was run on an Ubuntu 18.04 server with an Intel Xeon Benchmark Tardis-broken down into the different benchmark operations. The TardisBenchmark has to be processed. However, we also see a significant performance advantage in the retFirst scan in favour of the master branch. One reason related to this behaviour is the fact that the master branch always serves as the anchor point at the beginning of our version chain, therefore, no entries belonging to other branches have to be processed in this scenario. Better execution branch prediction results are one side effect of this approach, since we will only encounter entries belonging to the master branch. In general, processing version chains is low in cost since the difference between retFirst and retLatest is minor for all non master branches. We will see similar results regarding this difference in the TardisBenchmark with the Wikipedia workload.

6.3 TardisBenchmark
This section discusses the runtime of the four storage approaches broken down into the different benchmark operations. The TardisBenchmark was run on an Ubuntu 18.04 server with an Intel Xeon CPU E5-2660 v2 with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM. We used 2 TB of SSD storage to get the best performance out of the disk-based database systems SQLite and PostgreSQL. The benchmark was run single-threaded.

6.3.1 Insert. We inserted the Wikipedia dumps with full page edit history from 1 August, 2018 for pages 10 up to 2,087 using the four storage approaches presented. The pages were stored as snapshots or as differences in the database systems PostgreSQL version 10.5 (classical disk-oriented) and HyPer (in main-memory). The uncompressed XML dump file takes up to 76.3 GB of space. As the dump consists mostly of text, nearly the same amount of text was inserted into the database. To measure the performance of the operations for smaller datasets, we also ran our benchmark with only 3.7 GB of data (pages 30,227 to 30,303).

As we see in Figure 10, storing snapshots is always faster than computing the differences beforehand. That is valid for both PostgreSQL and HyPer, where the latter outperforms the disk-based database system, as no tuples have to be written to disk. LiteTree (see Figure 10a) shows the worst insert performance. When performing on the larger dataset (see Figure 10b), Git shows better runtime than PostgreSQL, but worse than HyPer, whereas SQLite is not capable of managing more than 1,024 branches at all. TardisDB—as optimised to this setting—was the fastest system.

6.3.2 Space Consumption. After inserting the pages, we analysed the space consumption in comparison to the original size of the two datasets (see Figure 13). The most space-consuming approaches—as no compression took place and the whole snapshots were copied—are LiteTree, HyPer-Snapshot and TardisDB-Snapshot.

LiteTree stores each page as a new database entry on each commit and creates no index structures. As it just copies the pages, its insertion time was one of the lowest ones, but also space consuming with more than the initial 3.7 GB. HyPer does not compress text-based data, so it needed the same amount of storage but in main-memory when storing each snapshot. One improvement would be to compress the articles manually before insertion, with the drawback of a higher runtime. Unexpectedly, PostgreSQL needed about 38 GB for the large dataset in both approaches (snapshot and

---

13https://dumps.wikimedia.org/enwiki/20180801/
Figure 10: Benchmark results for the insert operation: The left side shows the insertion of the small dataset, the right that of the larger. LiteTree was not capable of handling enough branches for the larger dataset.

Figure 11: Benchmark results for the retFirst operation with small (left) and larger (right) dataset. When pages are stored as snapshots, any version can be accessed immediately. When the differences between versions are stored instead, the first version has to be patched out of all deltas, which results in runtimes up to 50 times slower.

Figure 12: Benchmark results for retLatest operation with small (left) and larger (right) dataset. All storage approaches show about the same page retrieval time, as the latest version is always stored as a whole. The difference-based storage layout shows slightly better performance as no joins are performed.
While it performed the fastest after LiteTree were stored uncompressed. After we manually ran the garbage (see Figure 11b), the joins were eliminated. For the larger storage layout always showed the best performance except for the small load, the TardisDB-Snapshot initially needed about 38 GB—the same amount of memory as PostgreSQL. The benchmark used 1b6t for insertion, so the pages were stored uncompressed. After we manually ran the garbage collector git gc, the storage size shrank to 1.3 GB for the large dataset and 130 MB for the small one, so 1.7 % (3.51 % respectively) of the size of the uncompressed XML file. Here, as in all difference-based approaches, the bigger the dataset, the higher the compression rate. The size was reduced due to packing, the process of compressing the Git repository, implicitly called by the garbage collector. Packing the Git repository results in the lowest database size in our setup.

6.3.3 Retrieve First. Retrieving the first version works faster when the page content can be accessed as snapshots directly. This can be seen using either a main-memory or a classical database system as data storage for small (see Figure 11a) or large workloads (see Figure 11b). Git was always faster than the difference-based storage layout using its own delta compression. The performance of PostgreSQL decreased with the number of inserted elements: while it performed the fastest after LiteTree for the small load, the runtime decreased when the larger dataset was handled. LiteTree showed a good performance as every commit in a branch has an incrementally-growing number. To fetch the first version, we just specify it by article-branch. 1. Also in this setting, the architecture of TardisDB allowed the greatest number of retrieving operations per second, but this number decreases with higher workload, as a longer version chain has to be traversed.

6.3.4 Retrieve Latest. As we see in Figure 12, our difference-based storage layout always showed the best performance except for the small workload (see Figure 12a), where LiteTree performed the fastest. The reason that this outperformed the snapshot-based storage layout was that the joins were eliminated. For the larger workload (see Figure 12b), we see that the database systems outperformed Git by 20 % of its runtime. As TardisDB was tuned for fast scans on the latest tuples, it allowed the most read transactions independent of the workload size.

7 TAKE AWAY

Based on the evaluation, this section balances the benefits and downsides of storing snapshots or differences to propose the best versioning strategies.

7.1 Snapshot

Storing every snapshot of an article is a sufficient way of retrieving articles quickly. It can be implemented easily as a key-value storage without the need for external tools but condoning a high space consumption. To reduce the size, compression methods should be used, for example, PostgreSQL allows better storage strategies with EXTERNAL. Also, main-memory database systems would benefit from storage strategies as main-memory is limited.

7.2 Diff

When the differences are stored to the latest article, retrieving the latest one is quite fast, whereas for the first article, all deltas have to be patched. To reduce the number of deltas, we propose storing the whole snapshot every Nth revision to allow constant retrieval times (in $O(1)$ per operation). The factor $N$ controls the space requirement, together with $S$ as the total uncompressed size of all articles and $c_{avg}$ as the average size of a delta patch ($c_{avg} = 0.05$ for Wikipedia dumps). We obtain the following formula to approximate the storage size:

$$space(N, S, c_{avg}) = \frac{1}{N} \times S + \frac{N - 1}{N} \times c_{avg} \times S.$$  

We provide this idea as an experimental implementation as part of the TardisBenchmark. It outperforms retFirst and shows comparable runtime to retLatest but runs only in main-memory.

Another optimisation concerns the application of deltas. A delta looks like Replace from index X to Y with text T (implemented by string::replace). As a replace usually involves the copying and allocation of new memory, an improvement is collecting multiple deltas in order to allocate memory only once. For example, having two deltas, which each add a character to a string of size three, we can allocate a buffer of size five in advance to avoid string copying.

7.3 Solution

To summarise, we postulate the following requirements for a versioning system for text-based datasets:

- Reduce space consumption by compressing full articles
- Enable delta compression to reduce redundancy
- Constant retrieval times (with focus on the latest version)
- Database system guarantees as ACID properties, multi-user concurrency control and recovery
- SQL as a declarative programming language

The database systems’ guarantees, together with a declarative programming language, allows queries to be formulated with no concern for the actual implementation. This increases a program’s maintainability from a software engineering point of view.

Therefore, we propose integrating a modifier for existing text-like datatypes, such as TEXT VERSIONED, in the database system’s type logic. With this modifier, these datatypes behave similarly, but are optimised for texts that only change marginally between every
version. Such a modifier is adaptable for similar datatypes, such as XML or JSON, which would also benefit from compression.

8 CONCLUSION

This paper showed how to adapt versioning for database systems and evaluated the performance and space consumption of storing the differences between pages in contrast to storing the whole snapshots. We first developed a database schema, which was capable of managing versions over multiple relations, and described two tools—MusaeusDB and MusaeusSQL—to deal with the schema. To include versioning inside modern main-memory database systems, we adapted multi-version concurrency control for our prototype TardisDB that generated low-level machine code. In addition to timestamps for tuples and branches, the table scan operator verified in a version bitmap for each tuple whether that tuple is contained in the current branch. TardisDB performed best at retrieving the latest tuples, the setup for which it was designed. Our TardisBenchmark evaluated storing deltas against storing snapshots of pages based on the schema and data of MediaWiki. Our benchmark showed that it was indeed possible to shrink the amount of storage used by a factor of ten, while still offering comparable time when the pages are demanded. To further increase the retrieval time of prior versions, we investigated the internal storage layout and compression techniques of the engines used and proposed an algorithm that offers linear page retrieval time for the number of versions while consuming less space.

Overall, we showed versioning techniques that fit for database systems and proved the capability, with Wikipedia as a real world use case. To support further research on database versioning besides our proposed storage techniques, we provide our TardisBenchmark, which is capable of handling a flexible-sized workload.

Since the need does exist to version a single table’s attributes instead of the whole relation, we propose adding a versioning modifier for datatypes to database systems. In our scenario, only the page content has changed, while all meta information stayed the same. With such a modifier, the data could be stored independently, no further joins would be needed and no redundant information would be versioned.

ACKNOWLEDGEMENTS

This research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES