TardisDB: Extending SQL to Support Versioning

Maximilian E. Schüle
maximilian.schuele@tum.de
Technical University of Munich

Josef Schmeißer
josef.schmeisser@tum.de
Technical University of Munich

Thomas Blum
thomas.blum@tum.de
Technical University of Munich

Alfons Kemper
kemper@in.tum.de
Technical University of Munich

Thomas Neumann
neumann@in.tum.de
Technical University of Munich

ABSTRACT

Online encyclopaedias such as Wikipedia implement their own version control above database systems to manage multiple revisions of the same page. In contrast to temporal databases that restrict each tuple's validity to a time range, a version affects multiple tuples. To overcome the need for a separate version layer, we have created TardisDB, the first database system with incorporated data versioning across multiple relations.

This paper presents the interface for TardisDB with an extended SQL to manage and query data from different branches. We first give an overview of TardisDB's architecture that includes an extended table scan operator: a branch bitmap indicates a tuple's affiliation to a branch and a chain of tuples tracks the different versions. This is the first database system that combines chains for multiversion concurrency control with a bitmap for each branch to enable versioning. Afterwards, we describe our proposed SQL extension to create, query and modify tables across different, named branches. In our demonstration setup, we allow users to interactively create and edit branches and display the lineage of each branch.

CCS CONCEPTS

• **Information systems** → **Main memory engines**; • **Applied computing** → **Version control**.

KEYWORDS

Version Control, SQL

ACM Reference Format:
Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. 2021. TardisDB: Extending SQL to Support Versioning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3448016.3452767

1 INTRODUCTION

Online encyclopaedias such as Wikipedia rely on database systems to store the article's content but require dataset versioning to track the article's history. For this reason, MediaWiki, the software behind Wikipedia, implements its own version control to restore older versions on demand: one database table manages the pages' meta-information, another one contains their content with one entry per version and page [12]. Besides the increased complexity when querying the data, this approach wastes space by storing redundant information as each revision changes an article only marginally.

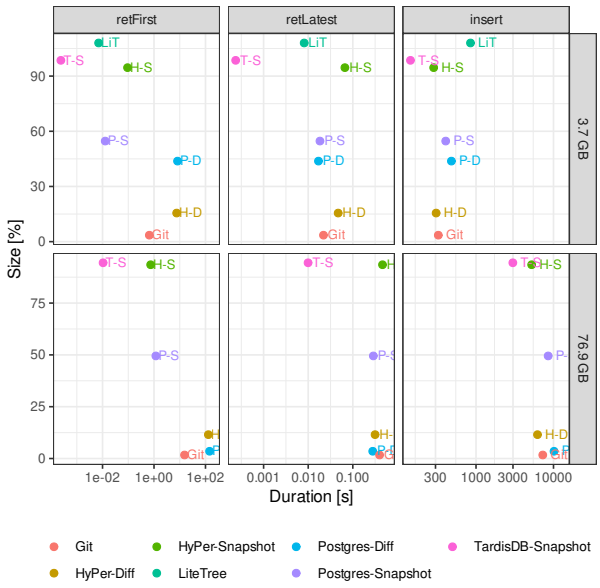

Figure 1: Performance when handling Wikipedia articles.

Another use-case for dataset versioning arises from the reproducibility needed to validate scientific experiments: To reproduce the same results, the dataset used should remain unchanged. However, a production database system must allow updates and insertions. Therefore, for reproduction, any published experiment must provide the corresponding data as a separate snapshot when the underlying data store does not support dataset versioning.

In summary, a dataset versioning tool should serve the purpose of a simple data storage system with version-based access as well as the needs for transactional data processing on multiple relations. So it has to fulfil reference key constraints across multiple tables within each version.

In previous studies [7, 13], we identified the following requirements for dataset versioning tools: They should reduce the space

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452767

consumption of the stored data, allow constant complexity when restoring a version, guarantee the ACID properties and provide a declarative query language such as SQL. We have benchmarked Git, HyPer, PostgreSQL, LiteTree and TardisDB using 3.7 GB and 76.9 GB of Wikipedia articles. The full page edit history[1] from August 1, 2018 (pages 10 to 2,087) was inserted either as a snapshot or compressed (*diff*), storing the changes as differences to the latest version only (backward deltas). We compared the duration for insertion (*insert*) and retrieval of the latest/first (*retLatest*/*retFirst*) version to the memory consumption (see Figure 1). Although TardisDB does not compress the data, it inserts and retrieves the articles faster than the competitors without integrated versioning. But a complete integration into a database system includes the extension of SQL. An extension of this declarative language would not only increase the acceptance of database systems for dataset versioning but also facilitate software engineering projects: When the data layer takes care of dataset versioning, other layers are not concerned in the actual implementation.

So this study solves the following problem: *How to integrate constraint-preserving dataset versioning within SQL?*

This paper demonstrates *TardisDB*, the first in-memory database system with incorporated SQL commands for database versioning. It is an open-source project[2] for a code-generating in-memory database system that produces LLVM code according to the producer-consumer model [11, 14]. Its table scan operator has been modified to produce only tuples for a selected branch. Therefore, each branch maintains a bitmap for every table, which denotes each tuple's visibility. Also, multiversion concurrency control is used to track each tuple's history. TardisDB is the first project that combines multiversion concurrency control with bitmaps for branching.

This study's contribution is the SQL extension for versioning, including an additional SQL statement for branch creation and an optional keyword as part of the from-clause to determine the branch. Named branches can be created using existing SQL keywords and fit seamlessly into declarative statements. We applied our changes to the Hyrise SQL parser [5], which allows our extension to be used for other systems as well. For demonstration purposes, we offer a web interface to interactively explore the proposed extension.

This demonstration paper is structured as follows: It first gives an overview of related studies on dataset versioning and explains the architecture of TardisDB. Then, we propose an extended SQL grammar, that is downward compatible with SQL-92, but allows branch creation and retrieving tables by version. Afterwards, we explain the demonstration setup, with an interactive web interface that allows users to create branches, examine lineages, query and modify the data.

2 RELATED WORK

For related work, we have to distinguish database versioning, which affects whole tables, from temporal databases, which restrict each tuple's validity to a time range. The first example of a temporal database system is *TQuel* [15]. The idea of time-restricted tuples is now part of the SQL:2011 standard [8]. Time-restriction affects each tuple individually, whereas a version comprises several.

[1]https://dumps.wikimedia.org/enwiki/20180801/
[2]https://github.com/tum-db/TardisDB

Table 1 lists tools for dataset versioning that are similar to database systems. One example with its versioning language VQL is the *DataHub* [1] platform whose flexible API allows applications such as an SQL interface to be built on top. The core part was highlighted as *Decibel* [10] with the language *VQuel* [4] to manage versions. These systems have in common that they build the application layer above the data storage rather than offering an extended relational algebra that supports versioning.

Using a relational database system, *OrpheusDB* [6] and our extension to support multiple tables for reference key constraints, *MusaeusDB*, provides a layer on top of database systems that projects versions onto database relations with a list of contained tuples per branch. As a fully integrated solution, *LiteTree*[3] is an extension to SQLite that allows access to and modification of any former database state using a version number within pragma statements. We also integrate versioning into a relational database system, but with a code-generating engine as the target, we adapt the table scan operator and multiversion concurrency control. For unstructured data, *Forkbase* [9] is a tamper-proof data storage system that allows named branches similar to Git and compresses data using similarity graphs. *R-Store* [2] is another key-value store based on Apache Cassandra, which also maintains bitmaps to indicate branches.

Table 1: Overview of data versioning tools.

System	Versioning	Querying	Data Model	Branching	Constraints
DataHub [1]	VQL	Thrift API	Relational	Numbers	No
Decibel [10]	VQuel	SQL	Relational	Numbers	No
ForkBase [9]	REST API	None	Key-Value	Names	No
LiteTree	Using Pragma	SQL	Relational	Numbers	No
MusaeusDB [13]	Bash Script	SQL	Relational	Numbers	Yes
OrpheusDB [6]	Python API	SQL	Relational	Numbers	No
RStore [2]	Java API	CQL	Key-Value	Numbers	No
TardisDB	SQL	SQL	Relational	Names	Yes

3 VERSIONING IN CODE-GENERATING DATABASE SYSTEMS

TardisDB follows the concept of a code-generating main-memory database system: It produces LLVM code according to the producer-consumer concept [11] where every operator demands the underlying ones recursively to generate code. Versioning now affects the table scan operator at the bottom of each operator tree: During compile-time, code for checking the bitmap as well as retrieving the correct tuple out of a version chain has to be generated. For every table and every branch, we maintain a bitmap, where a bit indicates the tuple's affiliation to a branch. On a table scan, a tuple will be produced only if the corresponding bit is set. When a branch is created, the parent bitmap is simply copied for every table. On an insert statement, all bitmaps are enlarged but the new bit is set for the affected branch only. On a delete statement, the corresponding bit is reset.

The bitmap indicates included tuples only but to track different versions of a tuple, we maintain version chains as used for multiversion concurrency control. This is the first time that multiversion concurrency is combined with branching to not only track a tuple's modification history but to provide access to a set of tuples that form one version. This approach tends to densely populated

[3]https://github.com/aergoio/litetree

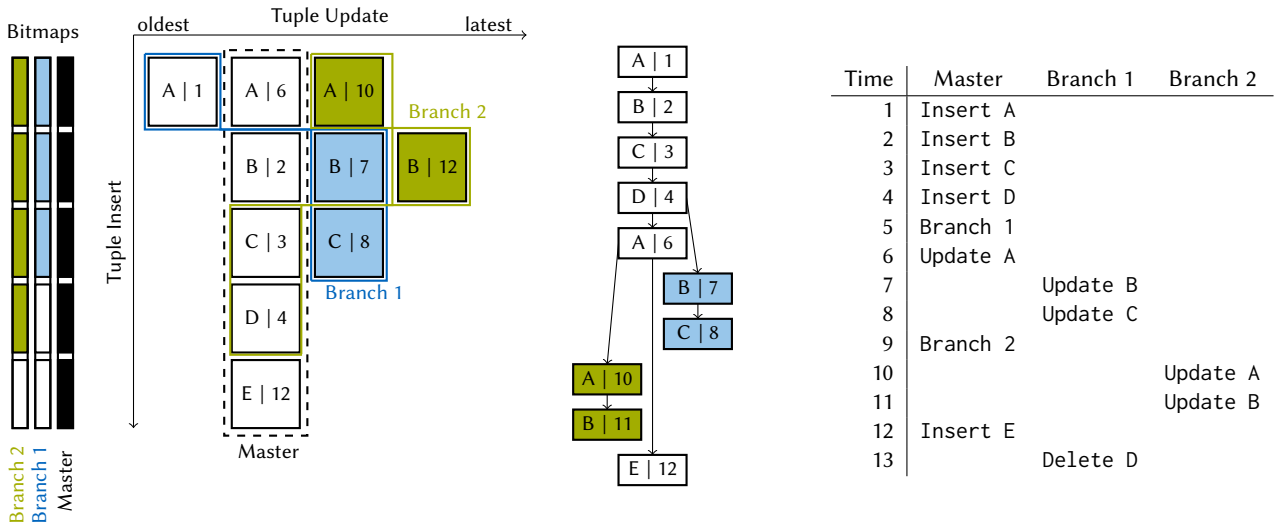

Figure 2: Left: Bitmaps for each branch (y-axis) mark contained tuples per table, whereas version chains track their history (x-axis), tuples of the master branch are stored in a column-oriented manner. Middle: Descendance tree. Right: Corresponding insert, update, delete and branching history.

bitmaps even on update-heavy workloads, which increases the total scan performance.

We define one prioritised branch, the master branch, whose tuples are stored in a regular column store and updated in-place. Each tuple is given two pointers to form a double-linked list of version entries and a timestamp, which is inherited from the branch that created the version.

Each branch, in turn, is given a timestamp at its creation, which is also used to retrieve the correct entry in the version chain: Formally, we define a predicate $active(t, b)$ for each entry t and branch b to indicate visible entries:

$$active(t,b) \Leftrightarrow crtd(b,t) \vee \bigvee_{p \in par(b)} active(t,p) \wedge ts(t) < ts(b).$$

An entry is visible for a branch when either the branch itself ($crtd(b,t)$) or a parent branch ($par(b)$) before furcation ($ts(t) < ts(b)$) has created the entry. The first active entry in the chain is the latest visible one and is returned.

Figure 2 visualises the history for five tuples on one master and two descending branches: Initially, four tuples were inserted before *Branch 1* is created with timestamp 5. The tuples for the master branch are stored column-wise (dashed line) and each contains a pointer to previous (left) and newer (right) versions. An update on the master branch changes the value in-place (A|6) and creates an entry for the previous version (A|1), an update on a descending branch just creates an entry on the right (B|7). When *Branch 1* requests tuple A, it receives A|1 as A|6 is only active within *Master* and *Branch 2*. Whereas two versions for tuple B are active within *Branch 1*, which is why it accesses the newer one (B|7). Instead of deleting a tuple, the corresponding bit is reset (D on *Branch 1*).

4 SQL EXTENSION

We now extend SQL (see Listing 1) to address the extended table scan operator. This requires a statement to create branches as well as an extension to specify the version for each table that is part of select, update, insert or delete statements. By default, regular SQL-92 queries without any adaptations are applied to the *master* branch to ensure downward compatibility.

```
CREATE TABLE users (id INT PRIMARY KEY, name TEXT);
CREATE TABLE things (id INT PRIMARY KEY, name TEXT, user INT
      REFERENCES users(id));
INSERT INTO users VALUES (1, 'Alice');
INSERT INTO things VALUES (21, 'printer', 1);
```

Listing 1: Example of tables created and filled with SQL.

To create a new branch, we add a statement to fork branches from existing ones (see Listing 2). This statement only expects the name of the parent branch and the created one. It does not require any further information as branches affect all tables to fulfil foreign key constraints. Creating a new branch is necessary to maintain access to a certain database state whenever this version should be preserved.

```
CREATE BRANCH mybranch FROM master;
```

Listing 2: Statement to create the branch *mybranch* from the parent *master*.

To query tables on the created branch, we propose the keyword VERSION behind each table that is part of an insert, select, update or delete query (see Listing 3).

```
INSERT INTO users VERSION mybranch VALUES (2, 'Bob');
UPDATE things VERSION mybranch SET user=2 WHERE id=21;
SELECT * FROM users VERSION mybranch;
```

Listing 3: Insert, update and select statements on tables of a certain branch using the VERSION keyword.

Although branches affect all tables, explicitly specifying the branch enables access to tables across different branches in one statement. This allows merging tables of different branches using ordinary SQL statements. For example, a full outer join allows

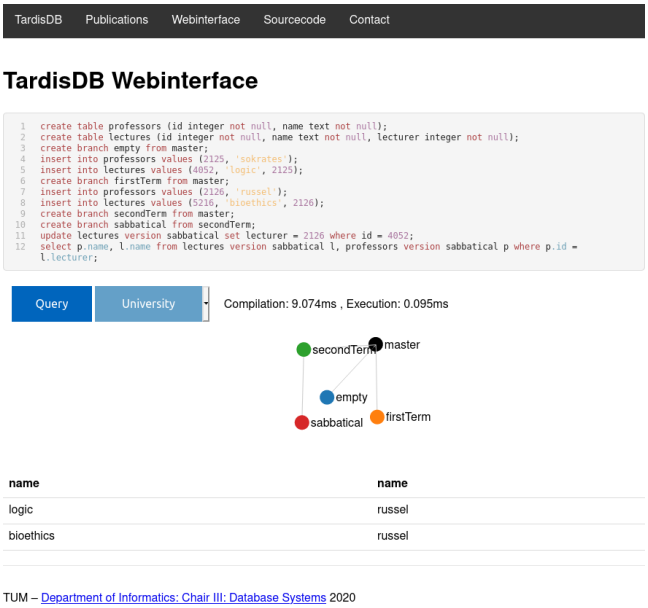

Figure 3: TardisDB web interface: An interface allows SQL queries including branch creation to be formulated. The chart in the middle displays the lineage of all available branches; the result table is shown at the bottom.

conflicting tuples sharing the same primary key to be identified (see Listing 4).

```
SELECT a.id, COALESCE(a.name, b.name)
FROM users VERSION master as a FULL OUTER JOIN
     users VERSION mybranch as b ON a.id=b.id
```

Listing 4: Merging tables.

Finally, after changes in a branch have been merged or become outdated, we propose a delete statement to free the allocated resources (see Listing 5). We propose a garbage collection [3] to remove versions that are not contained within a branch anymore.

```
DELETE BRANCH mybranch;
```

Listing 5: Branch deletion statement.

5 DEMONSTRATION SETUP

We have created an interactive web interface[4] to demonstrate extended SQL on TardisDB (see Figure 3). Within the web interface, we allow users to create tables, insert data and create branches. The lineage of created branches is visualised graphically. Of course, the SQL interface allows querying the data including joins over different branches. The result together with the query time is displayed afterwards. During the demonstration, we will start a TardisDB instance on a remote server that creates a new database instance for each client. This allows participants to try out the extended SQL on their own device even without physical participation. We will provide examples together with a selected CSV file as input data to demonstrate the performance of the available operators.

[4]http://tardis.db.in.tum.de

6 CONCLUSION

In this paper, we have demonstrated an extension of SQL to support versioning. Our extension supports named branches over multiple tables, which comprises a statement for branch creation and an auxiliary keyword after each table to determine the branch. We compiled SQL statements to operator plans based on an open-source parser, which allows integration into other software projects as well. Our target engine, TardisDB, was equipped with a modified table scan operator with bitmaps to indicate the affiliation of tuples to branches and a version chain to track their modification history. The extension did not slow down the read throughput on the master branch and retrieved other versions faster than comparable systems. With the developed web interface, we aimed to demonstrate the simplicity of extending SQL for versioning, which should also increase the acceptance of SQL for further tasks such as version control.

REFERENCES

[1] Anant P. Bhardwaj, Amol Deshpande, Aaron J. Elmore, David R. Karger, Sam Madden, Aditya G. Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. 2015. Collaborative Data Analytics with DataHub. *PVLDB* 8, 12 (2015), 1916–1919. http://www.vldb.org/pvldb/vol8/p1916-bhardwaj.pdf

[2] Souvik Bhattacherjee and Amol Deshpande. 2018. RStore: A Distributed Multi-Version Document Store. In *ICDE*. 389–400. https://doi.org/10.1109/ICDE.2018.00043

[3] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (2019), 128–141. https://doi.org/10.14778/3364324.3364328

[4] Amit Chavan, Silu Huang, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. Towards a Unified Query Language for Provenance and Versioning. In *TaPP*. USENIX Association. https://www.usenix.org/conference/tapp15/workshop-program/presentation/chavan

[5] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *EDBT*. 313–324. https://doi.org/10.5441/002/edbt.2019.28

[6] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya G. Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *PVLDB* 10, 10 (2017), 1130–1141. http://www.vldb.org/pvldb/vol10/p1130-huang.pdf

[7] Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. 2021. Umbra as a Time Machine: Adding a Versioning Type to SQL. In *BTW (LNI)*. Gesellschaft für Informatik, Bonn.

[8] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Record* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[9] Qian Lin, Kaiyuan Yang, Tien Tuan Anh Dinh, Qingchao Cai, Gang Chen, Beng Chin Ooi, Pingcheng Ruan, Sheng Wang, Zhongle Xie, Meihui Zhang, and Olafs Vandans. 2020. ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications. In *ICDE*. IEEE, 1718–1721. https://doi.org/10.1109/ICDE48307.2020.00153

[10] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. 2016. Decibel: The Relational Dataset Branching System. *PVLDB* 9, 9 (2016), 624–635. http://www.vldb.org/pvldb/vol9/p624-maddox.pdf

[11] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[12] Maximilian Schüle, Pascal Schliski, Thomas Hutzelmann, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. 2017. Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. *PVLDB* 10, 12 (2017), 1921–1924. https://doi.org/10.14778/3137765.3137809

[13] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *SSDBM*. ACM, 169–180. https://doi.org/10.1145/3335783.3335792

[14] Maximilian E. Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. The Power of SQL Lambda Functions. In *EDBT*. 534–537. https://doi.org/10.5441/002/edbt.2019.49

[15] Richard T. Snodgrass. 1987. The Temporal Query Language TQuel. *ACM Trans. Database Syst.* 12, 2 (1987), 247–298. https://doi.org/10.1145/22952.22956