Umbra as a Time Machine: Adding a Versioning Type to SQL

Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, Thomas Neumann
Dresden, Germany, May 10, 2021
Wikipedia: Version Control with Meta Tables

Maximilian E. Schüle (TUM) | Umbra as a Time Machine: Adding a Versioning Type to SQL
CREATE TABLE page (  
  page_id INT PRIMARY KEY,  
  page_title TEXT,  
  page_latest INT REFERENCES pagecontent (old_id)  
);  
CREATE TABLE revision (  
  rev_id INT PRIMARY KEY,  
  rev_page INT REFERENCES page (page_id),  
  rev_text_id INT REFERENCES pagecontent (old_id),  
  rev_parent_id INT,  
  rev_timestamp TIMESTAMP  
);  
CREATE TABLE pagecontent (  
  old_id INT PRIMARY KEY,  
  old_text TEXT  
);  


<table>
<thead>
<tr>
<th></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>
Background: Wikipedia Articles for Benchmarking

Monopedia (VLDB 2017)
- single main-memory database server (HyPer) sufficient for web scale applications (full English Wikipedia)
- Monopedia Benchmark: simulate load of Wikipedia

Monopedia: Staying Single is Good Enough

The HyPer Way for Web Scale Applications
Maximilian E. Schüle, Pascal M. N. Schliksi, Thomas Hutzelpeter, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, Thomas Neumann

Traffic Data Analysis: March 25-31, 2017: aggregated by week and hour to find out the traffic peaks and the time-dependent load distribution; more read requests during weekdays; nearly constant number of updates.

<table>
<thead>
<tr>
<th>Week</th>
<th>Reads</th>
<th>Writes</th>
<th>Reads/s</th>
<th>Writes/s</th>
<th>KiB/s</th>
<th>Reads</th>
<th>Writes</th>
<th>KiB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Week</td>
<td>7,677,200</td>
<td>0</td>
<td>966.9</td>
<td>171,624</td>
<td>31.2</td>
<td>821</td>
<td>125,107</td>
<td>33.0</td>
</tr>
<tr>
<td>2 Week</td>
<td>7,717,200</td>
<td>0</td>
<td>100.0</td>
<td>125,107</td>
<td>33.0</td>
<td>821</td>
<td>125,107</td>
<td>33.0</td>
</tr>
<tr>
<td>R1</td>
<td>12,000</td>
<td>0</td>
<td>322.9</td>
<td>499,654</td>
<td>32.4</td>
<td>801</td>
<td>322,107</td>
<td>32.4</td>
</tr>
<tr>
<td>R2</td>
<td>12,000</td>
<td>0</td>
<td>322.9</td>
<td>499,654</td>
<td>32.4</td>
<td>801</td>
<td>322,107</td>
<td>32.4</td>
</tr>
<tr>
<td>W1</td>
<td>12,000</td>
<td>0</td>
<td>322.9</td>
<td>499,654</td>
<td>32.4</td>
<td>801</td>
<td>322,107</td>
<td>32.4</td>
</tr>
<tr>
<td>W2</td>
<td>12,000</td>
<td>0</td>
<td>322.9</td>
<td>499,654</td>
<td>32.4</td>
<td>801</td>
<td>322,107</td>
<td>32.4</td>
</tr>
</tbody>
</table>

Versioning in Main-Memory Database Systems

Maximilian E. Schüle
maximilian.schuele@tum.de

Luke Karnowski
lukas.karnowski@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 constant-preserving version checks 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 type—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.

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 (diff) will reduce the amount of storage needed but will increase the retrieval time.
Space Compression

• Question: How to reduce the space consumption of Wikipedia articles within a database system?

Requirements for a Versioning System

compressing full articles  \[ \Delta \]  Efficient retrieval times (focus on latest version)  \[ \rightarrow \]  ACID/MVCC  SQL

Enable delta compression

CREATE TABLE page (page_id INT PRIMARY KEY, page_title TEXT, old_text DIFFTEXT);
Space Compression through \texttt{DiffText} Datatype

\begin{itemize}
  \item \textbf{Question:} How to reduce the space consumption of Wikipedia article within a database system?
  \item \textbf{Solution:} SQL datatype, that compresses text by avoiding redundancy
\end{itemize}

\begin{verbatim}
CREATE TABLE wikidiff (title text, content difftext);
INSERT INTO wikidiff (SELECT 'example', BUILD('first', 'first_version', 'second_version'));
SELECT GET_CURRENT_VERSION(difftext) FROM wikidiff;
\end{verbatim}
Umbra Integration
Umbra Integration

- Umbra: code-generating database system with in-memory performance
- offers SQL datatype for flexible sized strings
- DiffText datatype based on this datatype for variable length data
- latest version as a snapshot followed by backward deltas
- snapshots inbetween for efficient retrieval time

Figure: Chain of diffs, every third version a complete snapshot (bold).
Umbra Integration

- Umbra: code-generating database system with in-memory performance
- offers SQL datatype for flexible sized strings
- DiffText datatype based on this datatype for variable length data
- latest version as a snapshot followed by backward deltas
- snapshots inbetween for efficient retrieval time

\[ D_{T_2} \rightarrow T_1, D_{T_3} \rightarrow T_2, T_3, D_{T_5} \rightarrow T_4, T_5 \]

Figure: Chain of diffs, every third version a complete snapshot (bold).

```
struct DiffTextRepresentation {
    uint32_t currentOffset; // Offset of current version in data section
    uint32_t currentLength; // Length of current version
    uint32_t arraySize; // The size of the version pointer’s array
    uint16_t diffsToFullCount; // Counter of diffs until next full version
    struct {
        // Array of pairs, pointing into the data section
        uint32_t offset; // Offset of version in data section
        bool full; // Is this a full version?
        uint32_t patchStart; // Start of patch
        uint32_t patchEnd; // End of patch
    } versionPointers[];
    // Data section follows this struct immediately
};
```

Figure: Structure of a DiffText tuple.
SQL Integration
SQL Integration: Operations

- **BUILD**(\(T_1, \ldots, T_N\)) : creation of a DiffText object out of \(N\) versions.
- **APPEND**(\(D, T_1, \ldots, T_N\)) : appending \(N\) versions to an existing object.
- **SET_CURRENT_VERSION**(\(D, T\)) : adding a new version, equal to **APPEND**(\(D, T\)).
- **GET_VERSION_BY_ID**(\(D, N\)) : extract version \(N\).
- **GET_CURRENT_VERSION**(\(D\)) returns the latest version, equal to **GET_VERSION_BY_ID**(\(D, N\)) (#Versions=\(N\)).
- **EXPAND**(\(D, M, N\)) : unary database operator for extracting versions \(M\) to \(N\).
CREATE TABLE example (value DiffText); INSERT INTO example (SELECT BUILD(’first’));

<table>
<thead>
<tr>
<th>currentOffset</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLength</td>
<td>5</td>
</tr>
<tr>
<td>arraySize</td>
<td>0</td>
</tr>
<tr>
<td>diffsToFullCount</td>
<td>0</td>
</tr>
</tbody>
</table>

Data | First
---|---
CREATE TABLE example (value DiffText);
INSERT INTO example (SELECT BUILD('first'));
UPDATE example SET value=SET_CURRENT_VERSION(value,'first version');

<table>
<thead>
<tr>
<th>currentOffset</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLength</td>
<td>13</td>
</tr>
<tr>
<td>arraySize</td>
<td>1</td>
</tr>
<tr>
<td>diffsToFullCount</td>
<td>1</td>
</tr>
</tbody>
</table>

| offset | 0 |
| full   | false |
| patchStart | 5 |
| patchEnd  | 13 |

<table>
<thead>
<tr>
<th>Data</th>
<th>First</th>
<th>_Version</th>
</tr>
</thead>
</table>

Maximilian E. Schüle (TUM) | Umbra as a Time Machine: Adding a Versioning Type to SQL
SQL Integration: Example

CREATE TABLE example (value DiffText);
INSERT INTO example (SELECT BUILD('first'));
UPDATE example SET value=SET_CURRENT_VERSION(value,'first version');
UPDATE example SET value=SET_CURRENT_VERSION(value,'second version');

| currentOffset | 5 |
| currentLength | 14 |
| arraySize     | 2 |
| diffsToFullCount | 2 |

| offset | 0 |
| full   | false |
| patchStart | 5 |
| patchEnd  | 13 |

| offset | 0 |
| full   | false |
| patchStart | 0 |
| patchEnd  | 6 |

| Data         | FirstSecond_Version |
SQL Integration: Example

```
INSERT INTO example (SELECT BUILD('first', 'first_version', 'second_version'));
```

<table>
<thead>
<tr>
<th>currentOffset</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLength</td>
<td>14</td>
</tr>
<tr>
<td>arraySize</td>
<td>2</td>
</tr>
<tr>
<td>diffsToFullCount</td>
<td>2</td>
</tr>
<tr>
<td>offset</td>
<td>0</td>
</tr>
<tr>
<td>full</td>
<td><em>false</em></td>
</tr>
<tr>
<td>patchStart</td>
<td>5</td>
</tr>
<tr>
<td>patchEnd</td>
<td>13</td>
</tr>
<tr>
<td>offset</td>
<td>0</td>
</tr>
<tr>
<td>full</td>
<td><em>false</em></td>
</tr>
<tr>
<td>patchStart</td>
<td>0</td>
</tr>
<tr>
<td>patchEnd</td>
<td>6</td>
</tr>
<tr>
<td>Data</td>
<td>FirstSecond_Version 0123456789012345678</td>
</tr>
</tbody>
</table>

>SELECT GET_CURRENT_VERSION(value) FROM example;

>SELECT GET_VERSION_BY_ID(value,2) FROM example;

>SELECT GET_VERSION_BY_ID(value,1) FROM example;

Maximilian E. Schüle (TUM) | Umbra as a Time Machine: Adding a Versioning Type to SQL
SQL Integration: Example

```
INSERT INTO example (SELECT BUILD('first', 'first_version', 'second_version'));
```

<table>
<thead>
<tr>
<th>currentOffset</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLength</td>
<td>14</td>
</tr>
<tr>
<td>arraySize</td>
<td>2</td>
</tr>
<tr>
<td>diffsToFullCount</td>
<td>2</td>
</tr>
</tbody>
</table>

```
0 ⇒ Patch: [0,5)
false
0+5
6+5
```

```
SELECT GET_CURRENT_VERSION(value) FROM example;
SELECT GET_VERSION_BY_ID(value,2) FROM example;
SELECT GET_VERSION_BY_ID(value,1) FROM example;
```

```
First Sec
Second_Version
0123456789012345678
```
SQL Integration: Example

```
INSERT INTO example (SELECT BUILD('first', 'first_version', 'second_version'));
```

<table>
<thead>
<tr>
<th>currentOffset</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLength</td>
<td>14</td>
</tr>
<tr>
<td>arraySize</td>
<td>2</td>
</tr>
<tr>
<td>diffsToFullCount</td>
<td>2</td>
</tr>
</tbody>
</table>

Patch:
```
[0, 0]
```

```
>SELECT GET_CURRENT_VERSION(value) FROM example;
Second_Version
>SELECT GET_VERSION_BY_ID(value, 2) FROM example;
First_Version
>SELECT GET_VERSION_BY_ID(value, 1) FROM example;
First
```

<table>
<thead>
<tr>
<th>offset</th>
<th>0 ⇒ Patch: [0, 0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>13+6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>offset</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>false</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Data
```
0123456789012345678
```

Maximilian E. Schüle (TUM) | Umbra as a Time Machine: Adding a Versioning Type to SQL
Evaluation: Space Consumption

- **System**: Ubuntu 18.04 LTS, Intel Xeon CPU E5-2660 v2 processor, 2.20 GHz (20 cores), 256 GiB DDR4 RAM
- **Data**: Wikipedia dumps from 09/01/2019 (pages 971896 to 972009), Size of XML file: about 120 MB
- **Observation**: Good compression when storing every 20th version as a snapshot

**Figure**: Memory consumption depending on the frequency of stored snapshots.
Evaluation: $X=50$

Listing 1: Diff

```
INSERT INTO t (text) VALUES (BUILD(T1, ..., TN));
SELECT EXPAND(text, 1, N) from t;
```

Listing 2: Snapshot

```
INSERT INTO t (rev_id, text) VALUES (1, T1), ..., (N, TN);
SELECT text from t;
```

![Pie chart showing fractions for Snapshots, Header and Array, and Patches.]

```
Snapshots: 29.57%
Header and Array: 1.27%
Patches: 69.16%
```

Figure: Fractions.

![Bar chart showing time in seconds for Diff and Snapshot versions.]

```
Diff: Insert (Compile): 1, Insert (Execution): 1, Select (Compile): 1, Select (Execution): 1
Snapshot: Insert (Compile): 1, Insert (Execution): 1, Select (Compile): 1, Select (Execution): 1
```

Figure: Each version as a single snapshot or in one DiffText object.

- **Observation:** DiffText datatype faster for insertion and retrieval as less operations are required.
- Parsing time for text input part of compile time for insertion.

Maximilian E. Schüle (TUM) | Umbra as a Time Machine: Adding a Versioning Type to SQL
Conclusion and Future Work

CREATE TABLE wikidiff (title text, content difftext);
INSERT INTO wikidiff (SELECT 'example', BUILD('first', 'first_version', 'second_version'));
SELECT GET_CURRENT_VERSION(difftext) FROM wikidiff;

Conclusion
• SQL text data type for compression
• focus: retrieval times, not build time
• achieved compression rate of about 88 % (62 MB each snapshot within the database, 7.36 MB when storing the differences for \( X = 10000 \))

Future Work
• investigate on algorithms with stronger compression
• optimise build times
• store older versions on background memory
• combine with table versioning
Thank you for your attention!