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

Lukas Karnowski[1], Maximilian E. Schüle[2], Alfons Kemper[3], Thomas Neumann[4]

Abstract: Online encyclopaedias such as Wikipedia rely on incremental edits that change text strings marginally. To support text versioning inside of the Umbra database system, this study presents the implementation of a dedicated data type. This versioning data type is designed for maximal throughput as it stores the latest string as a whole and computes previous ones using backward diffs. Using this data type for Wikipedia articles, we achieve a compression rate of up to 11.9 % and outperform the traditional text data type, when storing each version as one tuple individually, by an order of magnitude.

1 Introduction

Version management of texts is still an important issue due to various use cases. The highlighted example is Wikipedia [Sc17], where people work decentrally on the creation of articles. In order to review their work, version management is mandatory, as it allows administrators to restore any previous version. As even versions of 2001—the founding year of Wikipedia—are accessible, an efficient storage of the data is necessary. Such a data storage should allow fast retrieval of previous versions, new versions to be inserted quickly and consume as little memory as possible.

Temporal databases such *TQuel* [Sn87] or as included in the SQL:2011 standard [KM12] restrict each tuple's validity to an added time range. In contrast, systems for relational dataset versioning such as *Decibel* [Ma16] lock on a higher granularity to track the history of whole tables. *VQuel* [Ch15], *OrpheusDB* [Hu17] and *LiteTree*[5] aim at combining SQL [Sc19] and versioning, but do not compress similar text strings. A stand-alone system that includes text compressing is *Forkbase* [Li20] but it is not interoperable with database systems.

```
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;
```

List. 1: Proposed data type `DiffText` for text versioning.

To measure the potential of compressing text strings, we have benchmarked storing strategies on popular relational database systems using the Wikipedia page edit history. This work

[1] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, lukas.karnowski@tum.de
[2] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, m.schuele@tum.de
[3] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[4] TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de
[5] `https://github.com/aergoio/litetree`

cc by sa

continues the study about versioning in main-memory database systems [SKS+19]: We propose a versioning data type, that can be used as an SQL attribute to store multiple versions of a text within one tuple (see List. 1). The developed data type for the Umbra database system [NF20] is presented in Section 2 by considering the diff algorithm, the memory layout and the implementation of the operations. Section 3 provides an evaluation of the data type's performance. Finally, Section 4 summarises the findings.

2 DiffText Data Type

In this section, we propose a *DiffText* data type to compress multiple versions of a text string as one database attribute within a tuple. The data type is based on the `BLOB` or `TEXT` data type that is available in many database systems to store byte sequences of any length. It thus inherits their properties with regard to the memory layout. This includes flexible size within a tuple to be enlarged as required, which is necessary when adding new versions. The data type is used as an SQL attribute: its values are overwritten on updates and copied for each occurrence as a column. This section presents the used algorithm for compressing strings, the data type's memory layout and necessary operations to retrieve versions out of a tuple.

2.1 Delta-Compression Algorithm

The DiffText data type applies delta compression to multiple versions of a string. It relies on difference-based versioning as it stores at least the latest version as a snapshot and restores the remaining ones using relative changes to the current version (*backward diffs*).

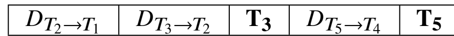
The idea is to access each byte of both versions only once. A function `find_diff()` determines the first and the last differing byte between two consecutive versions. First, both texts were compared from the beginning until the first differing byte has been found. The process is then repeated from the end of the texts. All bytes between these two boundaries found are called a *patch* and are part of the resulting diff even if they have bytes in common.

Example: The first step of the call `find_diff(aacbb, adddbb)` terminates after the second byte ($a \neq d$). The second step terminates after the third character from the back ($c \neq d$). The resulting diff is the string `ddd`, called patch, with the additional information that the second and third characters in the first text must be replaced. A complete diff thus consists of three parts, patch, start and end. The interval at which the patch must be applied is called `patchStart` and `patchEnd`.

When more than two versions exist, an order must be defined in which direction the patches will be applied. We decide in favour of *backward diffs*: The most current version is always available as a complete text, whereas older versions are stored as the difference to the version that was inserted afterwards.

Example: Assuming a newer version T_2 replaces the current one T_1. Having two similar text strings, the function `find_diff()` calculates the diff $D_{T_2 \to T_1} =: D_1$, so that T_1 can be reconstructed out of T_2 and D_1. The entire text of T_1 is then discarded and replaced by D_1. If another version T_3 is added, the diff $D_{T_3 \to T_2} =: D_2$ will be calculated and saved to replace T_2. If we want to reconstruct T_1, we will first apply D_2 to get T_2 and then apply D_1 to get T_1.

This process creates a chain of diffs that must be applied to restore older versions. Specifically, the number of involved patches increases with the number of inserted versions. For this reason, it is advisable to periodically save the complete version instead of calculating a diff. This allows constant access times in $O(1)$ to any version. Assuming that every third version should be complete and two additional versions T_4, T_5 are inserted, the chain of diffs would look like in Figure 1. Only two versions are complete and the remaining ones can be restored using diffs.



$D_{T_2 \to T_1}$	$D_{T_3 \to T_2}$	$\mathbf{T_3}$	$D_{T_5 \to T_4}$	$\mathbf{T_5}$

Fig. 1: Chain of diffs, with every third version as a complete snapshot (bold).

2.2 Memory Layout

To enable efficient operations later on, all versions of a text string are stored as one object. This leads us to the memory layout, which corresponds to the output of the presented algorithm out of patches and corresponding ranges. The actual patch is saved separately from the start and end of the diff. Figure 2 shows the schematic representation of the memory layout of the DiffText data type and Figure 3 the associated code.

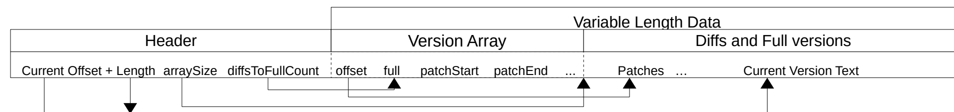


Fig. 2: Structure of a DiffText tuple.

```
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
};
```

Fig. 3: Source-code of the DiffText representation.

The layout starts with a header that indicates the size of the subsequent area. This variable-sized area contains all versions as diffs and is further divided into two parts: The first part contains a version array out of an offset, a flag `full`, and a range (`patchStart`, `patchEnd`). `patchStart` and `patchEnd` indicate the position that need to be changed in order to restore the previous version. The offset acts as a pointer to the last area in which the associated patch is located. Consequently, the last memory section is the concatenation of all patches and complete versions from which any version can be restored.

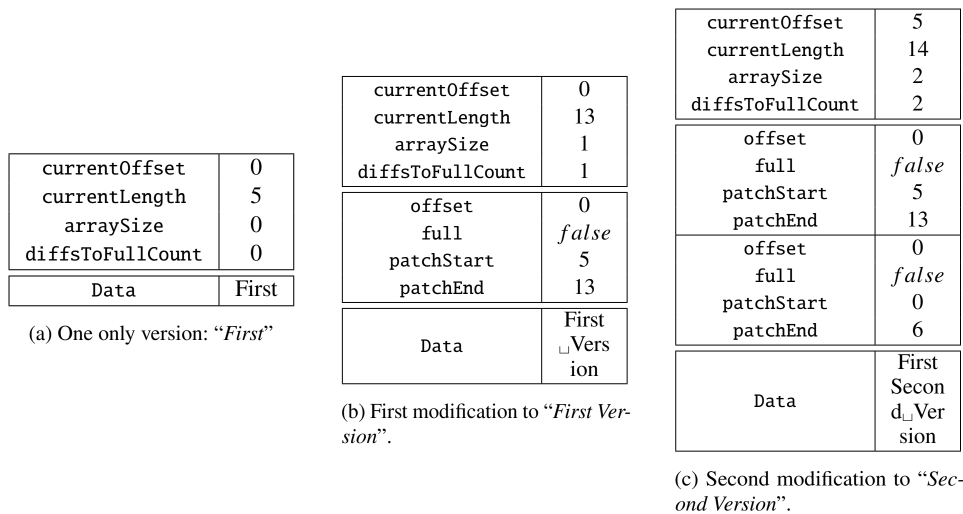
Since the latest version is always stored as a complete snapshot, the header contains an offset to the latest version in the data area in order to accelerate its access. The header also contains the current number of diffs that must be applied to restore a version (`diffsToFullCount`). Its value is incremented when a new version has been added. After reaching a predefined number, instead of calculating a patch, a complete snapshot will be saved, as presented in Section 2.1. This method ensures that each version can be extracted in $O(1)$. As the text's length is not stored, the tag `full` in the version array indicates whether the corresponding version has been stored as a complete snapshot instead of a patch.

For the offsets in the data area, 32 bit numbers have been used as Umbra's text-based data types are limited to 2^{32} bytes. This restrains the DiffText data type as all versions concatenated may not exceed a maximum size of 4 GiB. 16 bit was chosen for `diffsToFullVersion`, to avoid diff chains longer than 65536 as the runtime increases linearly with the number of patches.

Furthermore, only the offset is saved and the length of the patch is omitted. This is possible as the offset of the subsequent diff determines the end of the previous one. An exception is made for the current version, whose length is saved for fast retrieval.

2.3 Example for a DiffText object

For a better understanding of the memory layout, this section demonstrates the construction of a DiffText object by the following example: The initial version "*First*" will be changed to "*First Version*" by adding "*Version*". Then the current version is set to "*Second Version*". The resulting DiffText objects are listed in Figure 4.



(a) One only version: "*First*"

(b) First modification to "*First Version*".

(c) Second modification to "*Second Version*".

Fig. 4: States of a DiffText object when updating its entry with the following versions: (a) "*First*", (b) "*First Version*" and (c) "*Second Version*". The current version is reconstructed out of the text string in `Data` using `currentOffset` and `currentLength`.

Figure 4a shows the initial state with only one version. The version array is empty (`arraySize` = 0) and the only content in the variable-sized memory area is the current version "*First*".

After updating the entry, Figure 4b shows the second state with the version array containing one entry. Since backward diffs are used, this entry contains information on how to restore the previous version "*First Version*" from the current version "*First*". In this case, the content has to be cut off after "*First*". Accordingly, the coded patch in the version is a character string with a length of 0 (`offset` = 0). Since no length is stored in the version array, the length is implicitly calculated from the start of the subsequent version. In this case the following version is the current one, which is why the field `currentOffset` is considered. The length of the patch is therefore `currentOffset-versionPointers[0].offset=0`. The fields `patchStart` and `patchEnd` indicate at which point the patch must be inserted: In this case, the interval $[5, 13)$ corresponds to the added character string "*␣version*".

Figure 4c depicts the final state after inserting "*Second Version*". The version array now contains two entries: the first entry remains unchanged, whereas the second specifies how to restore "*First Version*" out of "*Second Version*". This is done by replacing the word "*Second*" with "*First*", so the patch must contain the latter character string. The interval $[0, 5)$ results from the `offset` entry in the array and `currentOffset` in the header, i.e. the first five characters in the data area ("First"). This patch is inserted in-between `patchStart` and `patchEnd` in the area $[0, 6)$ of the current version, which corresponds to the already mentioned replacement of the first word.

Also of interest is the field `diffsToFullCount`, which is equal to `arraySize` in our example. If more versions are inserted and `diffsToFullCount` reaches a predefined threshold value, a complete version will be saved, which is indicated by the tag `full` in the version array. `diffsToFullCount` is then reset to 0 and the process starts again.

2.4 Implementation of the Corresponding Operations

Since the data type was developed in Umbra, which currently does not support `UPDATE` operations, a copy of the previous state must be created for each operation. The data type supports the following functions:

- `BUILD(`$T_1, \ldots, T_N$`)` creates a DiffText object from a set of N versions. T_1 corresponds to the oldest version and T_N to the latest one. This can be used for recovery operations, for example, when creating backups out of bare text strings.
- `APPEND(`$D, T_1, \ldots, T_N$`)` is a generalisation of the `BUILD` operation. It expects a DiffText object D, to which the versions $T_{1 \ldots N}$ are appended.
- `SET_CURRENT_VERSION(`D`, `T`)` is a specialisation of `APPEND`, as it modifies a single version only, the standard operation for adding a new version.
- `GET_VERSION_BY_ID(`D, N`)` extracts version N from the given DiffText object. SQL is typically indexed starting with 1, with lower numbers indicating older versions and higher numbers corresponding to newer versions.
- `GET_CURRENT_VERSION(`D`)` returns the latest version. If the data type contains M versions in total, it is equivalent to `GET_VERSION_BY_ID(`D, M`)`. For performance reasons, a separate and optimised operation is offered to retrieve the latest version. The structure of the DiffText data type is designed to extract the latest version as quickly as possible. This will be discussed later in more detail.

In addition, `EXPAND(`D, M, N`)` is a unary database operator that extracts the versions within the interval $[M, N]$ out of a single DiffText object. It expects a relation with a DiffText column as input and returns $N - M + 1$ output tuples per input tuple. For performance reasons, newer versions appear first (the output order is $T_N, T_{N-1}, \ldots, T_M$).

This subsection presents the implementation of the previously presented operations for the DiffText data type.

2.4.1 Accessing Versions

Accessing an arbitrary version demands for the complete reconstructed text string. This is trivial for `GET_CURRENT_VERSION`, which is stored as a snapshot. In addition, its access

does not require to query the version array, only the offset and the length are read from the header. Furthermore, instead of allocating memory for the returned string, a view to the substring containing the snapshot is sufficient as return value.

The same optimisation will apply if the version requested by `GET_VERSION_BY_ID` is available as a snapshot (`full = true` in Figure 3). If this is not the case, a new buffer must be created for the return string. For performance reasons, the buffer size must be determined in advance. This is not trivial, since any diffs in-between might increase, decrease or leave the length of the resulting text unchanged. For this reason, `GET_VERSION_BY_ID` consists of 3 steps:

1. *Finding the next complete version.* This iterates from the requested version upwards through the version array until a complete snapshot is found. This could also be the latest version, this special case must be considered, since the most current version is not contained in the version array.
2. *Calculating the buffer size.* This requires again an iteration but in reverse order. In each step, the `patchStart` and `patchEnd` fields are used to calculate the resulting buffer size. The required buffer size is the maximum of all sizes found during all iterations.
3. *Applying patches.* In the last step, the version array is iterated downwards again and the corresponding diff is applied in each step. After this process, the requested version is available in the allocated buffer and ready to be returned.

This explains the separation into `patchStart`/`patchEnd` information and the actual patches within the memory layout: The first two steps do not require the actual patch, but only the meta information of each diff. This ensures optimal cache utilisation.

A further optimisation applies to `EXPAND`: Instead of iteratively calling `GET_VERSION_BY_ID` for each requested version, the patch is applied incrementally, starting with the last requested version. The implementation first calls `GET_VERSION_BY_ID` for the last requested version and then uses a function `getPreviousVersion(D,T)` to determine the predecessors. This implies that the order of the versions of the `EXPAND` operator is exactly counter-intuitive: starting with newer and ending with older versions. For performance reasons, however, this sequence is advantageous because only one step in the version array has to be carried out for each tuple output.

2.4.2 Creating a DiffText object

The trivial case when creating a DiffText object is with exactly one existing version. For this, the `currentLength` of the `DiffTextRepresentation` is set to the length of the single version. The remaining fields are initialised with 0, the version array is empty and the variable data area contains the current version only.

If more than one version exists, the resulting object will hold all information for their restoration. For this purpose, the diff is formed between two adjacent text strings by iterating once over all bytes and forming the patch between the current and the subsequent version. If the buffer already contains the text string to be inserted, no patches will be copied, but the corresponding offsets have to be saved. After all patches have been created, the texts are iterated again and the part relevant for the diff is copied into the data section of the newly created DiffText object. Thus `BUILD` consists of two phases (1) *Calculating the diffs* and (2) *copying the patches to the final buffer*.

`APPEND` is a generalisation of `BUILD`, because in addition to the new versions, an existing DiffText object is specified, to which the versions are appended. Apart from this, `APPEND` does not differ to `BUILD`, why it will not be discussed in more detail. The same applies to `SET_CURRENT_VERSION`, the specialisation of `APPEND`, which reuses the two phases mentioned above.

3 Evaluation

This section discusses the performance of the implemented DiffText data type. The Wikipedia dumps from 09/01/2019 were used as test data, specifically pages 971896 to 972009. The measurements have been conducted on 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.

The full dump has an uncompressed size of 119.9 MiB. First we evaluate the memory consumption after all available versions have been inserted. We add all versions of all pages in a DiffText object to better estimate the memory consumption. The result is shown in Figure 6. Instead of a patch, a complete snapshot will be stored every 50th version. This threshold, which restricts the chain length of patches, is referred to as X in the following.

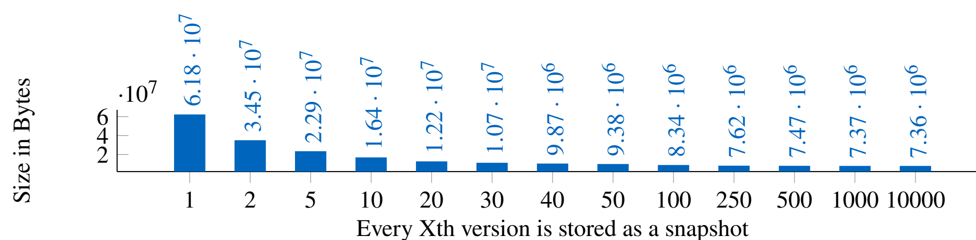


Fig. 5: Memory consumption depending on the frequency of stored snapshots.

The full size of the DiffText object is 8.9 MiB, which is a reduction down to 15.2 % of the original size. Figure 5 shows the total size of the DiffText object depending on the maximum chain length. Once a value of $X = 20$ has been exceeded, the memory consumption does not improve significantly the longer the chains become.

The best improvement achieve a chain length of $X = 10000$ with a reduction to 11.9 % of the original size. Compared to a value of $X = 50$, this means an improvement of only 3.3 percentage points condoning slower access to older versions. In [SKS+19] we achieved a compression to 5 %, which could not be reproduced in this work as the used Wikipedia dump includes less versions per article.

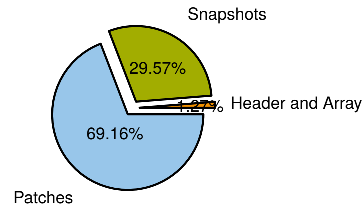


Fig. 6: Memory consumption with a complete snapshot every 50th version.

Let us now consider the runtime of the operations. A comparison with the normal `TEXT` data type is made by inserting the same versions of a text into a table with `TEXT` data as well as into a DiffText object. All revisions of one article are stored as one single DiffText tuple, while each snapshot is stored individually as a tuple. The comparison is therefore not representative as it compares different functions of the database system with one another. Nevertheless, the same amount of information is stored in both cases and a tenth of the memory is consumed in the case of the diff approach.

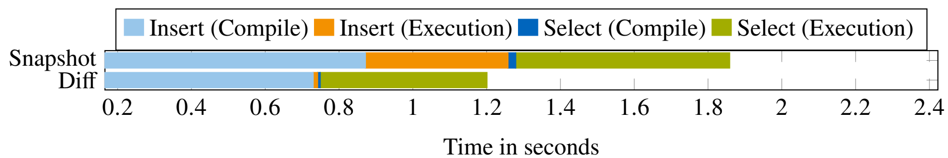


Fig. 7: Comparison: Storing each version as a single snapshot or in one DiffText object.

The following query inserts data into DiffText objects and retrieves text strings out of them:

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

List. 2: Benchmark queries using the `DiffText` data type.

The snapshots of all texts are inserted into the database as follows and then queried again:

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

List. 3: Benchmark queries using one tuple for each version.

Figure 7 compares the cumulative compilation and execution times of both approaches. The diff approach performs better than the snapshot approach in all metrics. The snapshot approach creates a tuple for each version and requires significantly more operations to insert the content.

4 Conclusion

In this work an implementation of a diff-based data type was presented, which is required for use cases like Wikipedia. New versions are created regularly, although older texts must still be accessible. The data type presented is implemented for the Umbra database system and is based on the normal text data type. The memory layout is designed for cache efficiency and consists of a header, a version array and the data area with patches and complete versions. The diff algorithm used is simple and can create diffs with just a single pass over the text.

The data type achieves a compression rate of up to 11.9 % of the original size for Wikipedia articles and is faster than the direct comparison with normal texts in both compilation and execution time. No other database system offers a similar data type so far, and research in this area is rather limited. Possible future optimisations for the data type include a larger storage capacity, storing older versions on background memory and a diff algorithm with stronger compression.

References

[Ch15] Chavan, A. et al.: Towards a Unified Query Language for Provenance and Versioning. In: TaPP. USENIX Association, 2015.

[Hu17] Huang, S. et al.: OrpheusDB: Bolt-on Versioning for Relational Databases. Proc. VLDB Endow. 10/10, pp. 1130–1141, 2017.

[KM12] Kulkarni, K. G.; Michels, J.-E.: Temporal features in SQL: 2011. SIGMOD Rec. 41/3, pp. 34–43, 2012.

[Li20] Lin, Q. et al.: ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications. In: ICDE. IEEE, pp. 1718–1721, 2020.

[Ma16] Maddox, M. et al.: Decibel: The Relational Dataset Branching System. Proc. VLDB Endow. 9/9, pp. 624–635, 2016.

[NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: CIDR. www.cidrdb.org, 2020.

[Sc17] Schüle, M. E. et al.: Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. Proc. VLDB Endow. 10/12, pp. 1921–1924, 2017.

[Sc19] Schüle, M. E. et al.: The Power of SQL Lambda Functions. In: EDBT. OpenProceedings.org, pp. 534–537, 2019.

[SKS+19] Schüle, M. E.; Karnowski, L.; Schmeißer, J., et al.: Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In: SSDBM. ACM, pp. 169–180, 2019.

[Sn87] Snodgrass, R. T.: The Temporal Query Language TQuel. ACM Trans. Database Syst. 12/2, pp. 247–298, 1987.