Time Travel in Column Stores

Martin Kaufmann [#*1], Amin Amiri Manjili [#*2], Stefan Hildenbrand [#3], Donald Kossmann [#4], Andreas Tonder [*5]

[#]*Systems Group*
ETH Zurich, Switzerland
[1]martinka@ethz.ch
[2]amamin@ethz.ch [3]stefanhi@ethz.ch [4]donaldk@ethz.ch

[*]*SAP AG*
Walldorf, Germany
[5]andreas.tonder@sap.com

***Abstract*— Column stores have been shown to outperform row stores significantly in a number of recent studies. This paper explores alternative approaches to extend column stores with versioning; i.e., the maintenance of historic data and time travel queries. On the one hand, adding versioning can actually simplify the design of a column store because it provides a solution for the implementation of updates, traditionally a weak point in the design of column stores. On the negative side, implementing a versioned column store is challenging because it imposes a two dimensional clustering problem: should the data be clustered by *row* or by *version*? This paper devises the details of three approaches: *clustering by row*, *clustering by version*, and a *hybrid clustering*. Performance experiments demonstrate that all three approaches outperform a (traditional) versioned row store and that the relative performance of the three versioned column store approaches depends on the query and update workload. Furthermore, the performance experiments analyze the time-space tradeoff that can be made in the implementation of versioned column stores.**

I. Introduction

In one of his last talks, Jim Gray postulated that "update in place" was dead [1]. Storage is becoming so abundant that it is cheaper to keep all data, rather than thinking about which data to delete. Instead of overwriting updated data, it is better to create a new version of the data.

There are a number of database products that support versioning. Correspondingly, these systems also support so-called *time travel* queries that allow to navigate to old versions of the data. Oracle has pioneered these ideas with its Flashback feature [2] which is integrated into the Oracle database product. Flashback extends SQL's `FROM` clause with an optional `AS OF` construct assigned to each table: `AS OF` specifies a version number or a timestamp that indicates which version of the table should be used. By default and in the absence of an `AS OF`, the latest version is accessed. In such a system, updates can only be applied to the latest version so that all historic versions are immutable. PostgreSQL had a similar feature based on the append-only design of the PostgreSQL storage manager [3]. ImmortalDB from Microsoft Research is a row store system that supports versioning and time travel queries [4].

So far, most work on versioning and time travel has been carried out in the context of a row store. Lately, however, it has become clear in numerous studies [5], [6] that *column stores* outperform row stores. In particular, column stores show superior performance for OLAP workloads. It turns out that support for time travel queries is particularly crucial in OLAP applications. For example, an analyst might be interested in the value of his portfolio today if he had left it unchanged since the beginning of the financial crisis in September 2008. This query involves a time travel to the state of the portfolio as of September 2008 and a reassessment of the value of that portfolio with current prices and stock quotes.

This paper presents alternative approaches to implement versioning and time travel queries in a main memory column store. The work was motivated by the time travel feature of an in-memory column store database system by SAP [7] which is designed to accelerate OLAP queries. The goal of this work was to find the best design for the time travel component of this system.

Implementing versioning and time travel in a column store is not trivial. The state-of-the-art implementation of versioning in row stores is based on chaining the versions of a record using pointers [4]. If versions are held in the granularity of individual fields as part of a column store, then the storage overhead of keeping such pointers can be prohibitive. Furthermore, a lot of optimizations carried out for column stores are based on a predictable sequential access pattern during the processing of the data; this optimization may become less effective if pointers are chased. Another issue is the organization of the column store. Typically, column stores are clustered by *row-ID* in order to make inner joins of columns of the same table fast. The question is how to store versioning information (i.e., *version-IDs*) and whether other clustering schemes become more attractive for time travel queries.

The main contribution of this paper is to study alternative approaches to implement versioning and time travel in main memory column stores. The described memory layouts differ in the way they encode versioning information and how they cluster the data. The first approach clusters by *row* (as in traditional column stores). The second approach clusters by *version-ID*. The third approach is a *hybrid* between the first two approaches. For each layout the basic data structures, query processing and update algorithms are shown. Furthermore, this paper presents the results of a comprehensive performance study that assesses the tradeoffs of the alternative

approaches and compares them to a state-of-the-art row store implementation. These experiments also give insight into the fundamental space-time tradeoffs of versioned column stores.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 presents use cases which are relevant for accessing historic data. Section 4 sketches which update granularities can be implemented in a column store. Sections 5-7 describe the three alternative approaches to implement time travel in column stores. Section 8 discusses the results of the performance experiments. Section 9 contains conclusions and possible avenues for future work. The Appendix gives a more formal description of the algorithms and presents additional experiments.

II. Related Work

Temporal and versioned databases have been subject to extensive research. A survey on the fundamental work on temporal databases is given by [8]. More recent related work and an overview of indexing techniques on temporal databases is presented in [9].

Management of historic data has been implemented in well known DBMS: [3] describes the implementation of an archive in *PostgreSQL*. Recent versions of *Oracle* support a similar feature called Flashback [2], [10] that allows going back in time. We adopt some of Oracles query language extensions to present our examples. The Flashback feature comes in different variants: a) short time row level restore using UNDO information, b) restoring deleted tables using a recycle bin and c) restoring a state of the whole database by storing before images of entire data blocks [11]. The newest variant of Flashback introduced in 11g called Flashback Data Archive stores the entire data augmented with the required meta-data in a dedicated archive using background processes.

Another database system that manages historical data is *ImmortalDB* [4], [9], [12]. It is built into a row-oriented system and timestamps the data in the granularity of records. Their chained representation of records (i.e. every record has a pointer to the next older version of itself) does not work in column stores as the overhead of pointers is significantly higher. We solved this problem in the *clustered by row* approach by chaining blocks of versions to keep the overhead low and to increase locality.

In the area of processing historical data there have been surveys like [13], [14] which present the foundations of temporal data management and access methods. As in [13] we will discuss physical layouts where the data is clustered by *row* (called key-only in [13]), by *version-ID* (called time-only) and one approach that combines the two (called time-key).

We investigate the area of processing historical data in the context of column stores: The idea of storing data in columns instead of rows dates back to [15]. The advantage is clear: Only the required columns are brought to the CPU via the memory hierarchy. However, the data needs to be reconstructed from the different columns in order to return the resulting row. This can usually be done quite efficiently if the data in the different columns is stored in the same order. Adding historical data to a column store imposes the following design problem: either we supplement the storage with data that is not required to answer the query (if we insert an entry in every column for each update) or we cannot have simple offset access to all columns as they might have received a different number of updates. This leads to the question to what extent versioning affects the advantages of a column store. To the best of our knowledge, this question has not been answered yet. *C-Store* has support for versioning using the multi version storage that enables snapshot isolation and the concept of a writable and read-optimized store [16]. However, this feature is limited in C-Store to short-term time travel for the implementation of snapshot isolation. C-Store does not support time travel queries.

Vertica [17] is a commercial in-memory column-store database system which (similmar to C-Store) provides a limited support for versioning by means of snapshot isolation.

Furthermore, the high similarity of adjacently located data in a column store can be exploited for compression which both reduces memory-consumption and improves query performance. An overview of different compression schemes in column stores is presented in [18].

III. Use Cases

In this section we describe the use cases we are considering for designing our implementation. There are basically two dimensions relevant to a relation that contains historic data: the time dimension (i.e. slice the relation to show state at a given point in time) and the row dimension (i.e. slice the relation to show the changes made to a certain row) as shown in Figure 1. In addition, combinations and aggregations are possible. The data can be clustered along at most one of these two dimensions. Depending on this decision, different costs have to be paid for different access patterns. The use cases presented in this section are selected to expose these tradeoffs.

In general, a *version-ID* represents a unique transaction time stamp in the database. For simplicity, we will not distinguish a *version-ID* from a date in real-word in the remainder of this paper. We will explain the use cases and our proposed memory layouts within examples based on versionized tables from TPC-H schema.

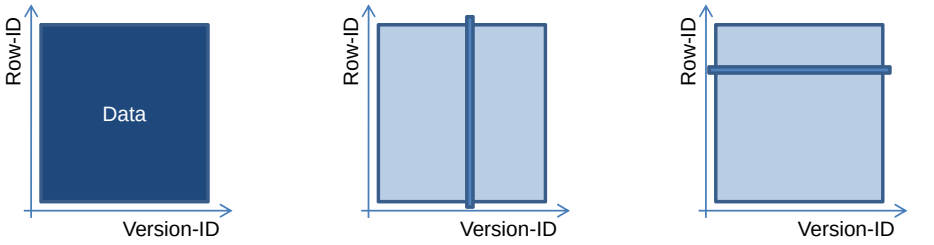
A. Time Travel

One application of historic data is the possibility to travel in time, meaning that the recording of historic data enables the user to see the database at a certain point in time. An example query could be: "What was the maximum ordered quantity in all *lineitems* at the end of last year?" This can be formulated as a SQL query:

```
SELECT MAX(l_quantity)
FROM lineitem
AS OF '2011-12-31'
```

B. Evolution of Data (Audit)

The other application of historic data slices the data along the other dimension, meaning that we query the changes of a specific value over time. An example for such a query is "What



(a) Whole Database (b) Slice at a Certain Point in Time (c) Evolution of a Certain Row

Fig. 1: Different Dimensions of a Relation with Historic Data

was the maximum quantity of a specific *lineitem* over the last five years?" This type of query is important to satisfy audit requirements (e.g. showing that the data was always within a certain range). The SQL syntax is along the lines of the Oracle *Flashback* [2] syntax. `VERSIONS` returns a row for each version of the specified data item:

```
SELECT MAX(l_quantity)
FROM lineitem
VERSIONS BETWEEN '2005-01-01' AND '2009-12-31'
WHERE l_linenumber='3' AND l_orderkey = '1'
```

Where $linenumber$ and $orderkey$ are compound primary key for *lineitem*. Note that some aggregations such as *average* are non-trivial to implement in the presence of time travel.

C. Record Reconstruction

Since accessing multiple attributes is different in row and column stores, we consider a query which returns the value of different attributes of a table at a certain time. In addition, a condition is defined. Thus, only a subset of all rows which existed in the database at that time is retrieved. The following SQL code is an example of such a query:

```
SELECT availqty, supplycost
FROM partsupp AS OF '2010-09-24'
WHERE suppkey < 10
```

D. Processing Inserts and Updates

There are three relevant additional use cases which are related to changing the information that is stored in the database. An insert operation adds an additional record (e.g. a new *lineitem* record). An update modifies an existing record and adds a new version without removing the old information from the storage. The delete operation is special in this scenario since the database is required to keep track of the history of the deleted rows in order to answer queries about the past consistently.

With the objective of supporting the use cases shown in this section, Sections V, VI and VII present different approaches to store a table in main memory with versioning-support in the granularity of a single column (attribute). Our design space for the memory layouts contains several dimensions; First, the data can be clustered either by row or by version. Second, replication of data improves query response time, but introduces a tradeoff between query execution time, update costs and memory consumption. Third, depending on the storage layout, different compression methods can be applied. In addition, dictionary encoding [18] and dictionary compression [19] are general compression methods which work both in row and column stores. Compression not only reduces the consumed amount of memory but also improves the execution time of queries which are executed over compressed data [18]. Furthermore, in case of archiving (moving old versions to harddisk), compressed data reduces the required space on harddisk and increases the speed of transferring data from main memory to disk and vice versa.

IV. Update Granularity

This section investigates how updates are applied to a table which contains several columns. As in a versionized table an update is implemented as an insert of a new version, the question is whether the entire affected row should be stored as a new version or only the modified attributes should be preserved.

A. Asynchronous Columns

For this approach, updates are only applied to the columns where the value has changed. Thus, the relative position of values for a given row and version is independent in different columns. For example, if in a row of the customer table only the address is updated, a new version is only written to the address column, but the other columns are not affected. The *asynchronous columns* approach is described in [20] and referred to as Temporal Decomposition Storage Model (TDSM).

The advantage of *asynchronous columns* is that updates can be executed very fast. In addition, memory consumption is efficient because no data has to be replicated. Read operations are fast for single columns.

On the negative side, performance of tuple reconstruction decreases with the number of columns which have to be joined. Since different columns have different sizes (due to different number of updates modifying them), it is not trivial how to efficiently find the corresponding value for a row in all columns.

B. Synchronous Columns

In this approach, each version of a row is stored at the same relative position *synchronously* in all columns. In the case

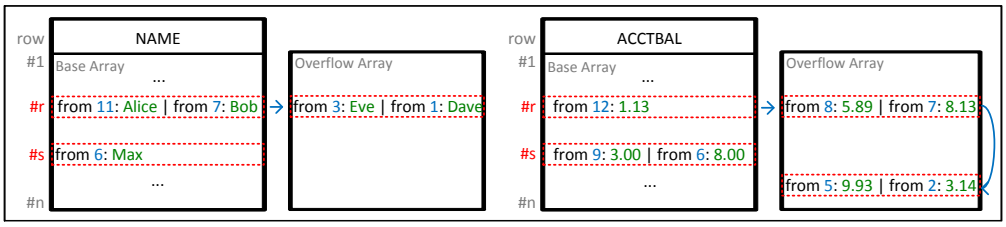


Fig. 2: Clustering by Row with 2 Versions per Row Segment

of an update, the previous value is replicated for unchanged columns and therefore the relative position of a value for a given row and version is identical in all columns which results in an efficient tuple reconstruction. The *synchronous columns* approach has been chosen in most commercial systems so far.

Very fast record reconstruction is the main advantage of synchronized columns. This benefit can be achieved by preserving the same relative position of all values for a given row and version in all columns. In addition, the *version-ID* has to be stored only once for one tuple because all columns are always updated at the same time.

On the negative side, *synchronous columns* lead to an increased update execution time and memory consumption due to replication of data. However, compression can be applied which benefits from the similarity of adjacent values in memory.

For the following presentation of three alternative memory-layouts, we will focus on the *asychronous column* approach.

V. Clustering by Row

This section introduces a memory layout in which the data is clustered by row-ID.

A. Storage Layout

In the *cluster by row* approach, for each row in a column space for a fixed number of versions is reserved. The memory layout contains a *base array* of *segments*. Each position in the *base array* corresponds to a row in the table. A *segment* contains $width_{row}$ pairs of (val_{im}, ver_m) as a payload rather than an atomic value like in a tradition column store. val_{im} is the value of row i which has been valid since version ver_m. If the number of updates of one row in the *base array* exceeds $width_{row}$, the data of the *segment* is copied to the next available position in an *overflow array* and a reference is stored. Within this *overflow array*, the *segments* of each row are chained and referenced by their array position.

In the example shown in Figure 2 we consider a versionized `customer` table with two attributes. If the account balance of customer s decreases to '$3.00' at version number '9' a new (val_{sm}, ver_m) pair with val_{sm}='3.00' and ver_m='9' is prepended to the segment of this customer. The former value is moved to the next available position in this *segment*.

B. Query and Update Processing

In this subsection we describe the operators needed to support the use cases which have been introduced in Section III-D. We show how these operators can be implemented efficiently for this layout. First, we describe operators to alter the stored data. Next, we will continue with operators to retrieve data from memory.

Insert. For insertion of a new row to a column, we append a *segment* to the *base array*. If the current number of *segments* exceeds the maximum number of rows $MaxSize_{byrow}$, space for a new column with size $2 * MaxSize_{byrow}$ has to be allocated and the data from the old column is copied. Next, a new *segment* is appended and the new (val_{im}, ver_m) pair is written to the leftmost position of the *segment*.

Update. As shown in the motivation of this paper already, in temporal tables old versions are never modified. Therefore, an update operation in a temporal database can be translated to inserting a new version.

If there is space for a new version in the corresponding *segment* of a row, the previous (val, ver) pair is moved from the leftmost position to the next unoccupied position within the *segment* and the new pair is written to the leftmost position. By this means, shifting all previous versions can be prevented. If the *segment* is full, the content of the *segment* has to be copied to an available position in the *overflow array*. The position within the *overflow array* in terms is used as a reference to chain the *segments*. Thus, references to previous *segments* never have to be updated because the position in the *overflow array* remains unchanged. Again, allocated space for the *overflow array* is doubled if maximum size $MaxOv$ is exceeded.

In the *cluster by row* layout, the implementation of *synchronous columns* is more difficult because for referencing (val, ver) pairs it is necessary to add the information if the pair is located in the *base* or *overflow array*.

Delete Operation. For simplification, we assume that the ID of a deleted row never is inserted again. In order to keep track of deleted rows, we choose a similar approach as presented in [16]. We introduce a bitmap in which a $true$ bit at position i indicates that row i has been deleted. In this case the last entry is a dummy update to keep track of the version in which this row has been deleted.

Next, for efficiently accessing data in memory, we define a set of scan operators retrieving values which fulfill a given condition. Optimal performance of each scan operator can

be achieved by exploiting the clustering characteristics of the underlying layout. In the following, we describe how the value for a specific version can be retrieved. We will continue with describing how an aggregation over a subset of versions for a row can be implemented.

Select Value for a Given Version and Row-ID. This operator retrieves a single value for a given row i and version ver. In this layout, the data is primarily clustered by row and secondarily clustered by version. Thus, the position of the row has to be determined in the array of *segments* first, which can easily be achieved by a simple array-lookup based on row-ID i. In the second step, the value which is valid for a given version ver has to be retrieved by sequentially scanning the (val_{im}, ver_m) pairs. *Segments* can be skipped by looking at the smallest pair per *segment* first. A value is valid for ver, iff $ver_m \leq ver$ and no other ver_q exists with $ver_m < v_q \leq v$. A pseudo code representation for this algorithm is given in Appendix A.

Select Value for a Given Version and a Group of Rows. This operator repeats the above mentioned operation for all rows in a given group.

Aggregation over a Version-Interval for a Single Row. This operation reads the values of a given row for all versions in a time interval $[ver_{start}, ver_{stop}]$ and calculates a single aggregated value. In a first step, the *row segment* for the given row-ID has to be located. Secondly, all values for versions within the time interval have to be read in the *segment* to calculate the aggregated result. For this purpose, all *segments* connected to this row in the *overflow array* have to be traversed successively. The calculation can be stopped when it holds $ver < ver_{stop}$.

C. Uncompressed Memory Consumption

Let $size(T)$ be the size of data type T (e.g. 4 bytes for Integer), $size(ver)$ the size of the version number (e.g. 8 bytes for Long), $size(pos)$ (e.g. 4 bytes for Unsigned Integer) the size of position of the next segment in array. Then the size of a *segment* is given by

$$size_{segment} = width_{row} * (size(ver) + size(T)) + size(pos)$$

Correspondingly, the total size of a column with $MaxSize_{byrow}$ rows and $MaxOv$ overflow segments is

$$size_{total,byrow} = (MaxSize_{byrow} + MaxOv) * size_{segment}$$

The most important parameter for this layout is the width of a segment $width_{row}$. On the one hand, a larger number of versions per *segment* provides faster access to historic data because less jumps in memory are required. On the other hand, a lot of memory is wasted if only a few rows are updated frequently. For the experiments, 10 versions per *segment* were chosen as a reasonable compromise.

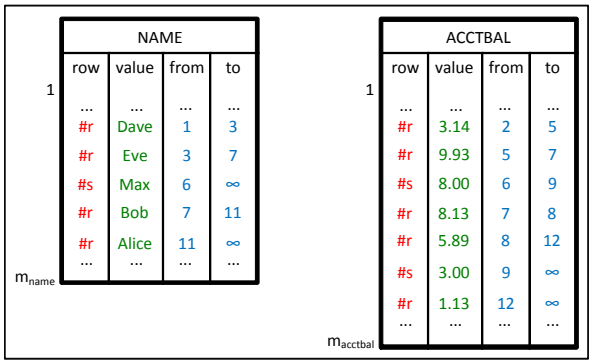


Fig. 3: Clustering by Version

D. Archiving

An archive allows storing parts of the table on harddisk, e.g. in case not all data fits in main memory. For creating an archive, a version $ver_{archive}$ is chosen as a threshold. All versions which are older than $ver_{archive}$ are stored on harddisk, newer versions are kept in main-memory. Archiving can be implemented for the *cluster by row* approach by moving all *segments* to harddisk for which the validity interval of all (val, ver) pairs is strictly smaller than $ver_{archive}$. Yet, the segment containing the value valid at $ver_{archive}$ must reside in main memory to reconstruct the value for this version without accessing the harddisk. The space of the released *segments* within the *overflow array* can be re-used for future *overflow segments*.

E. Compression

The *cluster by row* layout is a compact representation of updates per row. Therefore it is hard to achieve an additional reduction of memory consumption. However, dictionary encoding [18] and dictionary compression [19] can be applied to exploit the characteristics of the stored values.

F. Discussion

The *clustering by row* layout performs best for queries which access a large number of versions of the same row. This is the case for the evolution of data use case in subsection III-B. On the other hand, it is expensive to retrieve a very early version of a row if it has been updated many times because a large number of positions of overflow pages has to be accessed. Memory consumption is optimal only if all *segments* are fully occupied which is the case when the number of updates per row corresponds to the width of a *segment*. A lot of space is wasted if rows are never updated.

VI. Clustering by Version

In this layout the data is clustered by insertion-order, i.e., by *version-ID*.

A. Storage Layout

In the *clustering by version* approach visualized by Figure 3, for each version of a row four values are stored in an array: The row-ID i, the value val and a version interval

given by the version ver_{from} for which this value becomes valid and the version ver_{to} when it is invalidated. The version interval simplifies determining if a value is valid for a given version without having to scan all data to check if it has been invalidated within another update.

For example, the fact that the customer with row-ID #r had a balance of '$8.13' from '7' to '8' can be represented by (#r, '8.13', '7', '8').

B. Query and Update Processing

Insert. In the *clustering by version* approach, if a new tuple with row-ID i is inserted at version ver the tuple (i, val, ver, ∞) is appended to the array. If number of tuples in the column exceeds its maximum size $MaxSize_{byversion}$, space is doubled and values are copied as in the previous approach.

Update. As we have to store a version interval for each update, the end interval of the previous version of this row has to be set first. Finding the latest version can be done in constant time by looking up the position in a *latest version array*. This array of size $count_{row}$ stores the position of the latest tuple for each row-ID. An alternative to the *latest version array* is a backwards-scan to retrieve the latest value which is valid for a given row-ID.

As a next step, the new version is appended similarly to the insert operation in the previous section.

The *cluster by version* layout can support both the *asynchronous* and *synchronous columns* update granularities introduced in Section IV because it is possible to reference a tuple efficiently by its array position.

Delete Operation. We give two alternative implementations of the delete operation.

First, the latest version can be invalidated with ver_{to} being set to the deletion time and no new tuple being inserted. Second, a bitmap marking deleted rows can be kept within the *latest version array*.

Select Value for a Given Version and Row-ID. As the data is clustered by version in this layout, the operator is implemented as a scan with the version as the primary search criteria. In a first step, the position has to be found for which the values are valid with respect to the given version `v`. This is the case when it holds $ver_{from} \leq ver \leq ver_{to}$. Next, for each valid tuple the ID has to be compared to the given *row-ID* and the value is read and returned as a result when the ID matches. Note that a backward scan would be more efficient if `v` is closer to the latest version. A pseudo code representation of this algorithm is given in Appendix B.

The latest version can be found more efficiently with the first occurrence of a given row-ID i in a backwards-scan. Alternatively, the position of the latest value for each row-ID can be retrieved in constant time from the *latest version array* without having to scan the whole column.

Select Value for a Given Version and a Group of Rows. In contrast to the previous layout, in this layout there is no need to repeat the above operation for each row. One scan is enough to retrieve values for a group of rows.

Aggregation over a Version-Interval for a Single Row. This operation has to be implemented with a linear table scan. For each tuple, the ID is compared to the given row-ID i. If the IDs are equal, the time intervals are compared and the corresponding value is read and the aggregation can be calculated. As the data is sorted by ver_{from}, the calculation can by aborted when it holds $ver_{from} > ver_{stop}$.

C. Uncompressed Memory Consumption

For this memory layout, the size of one $(rowID, val, ver_{from}, ver_{to})$ tuple can be calculated by

$$size_{tuple} = size(rowID) + size(T) + 2 * size(ver)$$

The total size of the column is

$$size_{total,byversion} = MaxSize_{byversion} * size_{tuple}$$

The total memory consumption $size_{total,byrow}$ of the *cluster by row* approach is higher than $size_{total,byversion}$ if a lot of *segments* are left unoccupied. This depends on the workload and number of updates for each row.

D. Archiving

In order to archive previous versions on harddisk which are older than $ver_{archive}$, a scan of the column is necessary. A tuple can be moved from the column to disk if its validity interval is completely before $ver_{archive}$ which is fulfilled when it holds $ver_{to} < ver_{archive}$. After transferring the old tuples to the harddisk, the column has to be rewritten in order to free the memory of the tuples which have been moved to the archive.

E. Compression

In this layout the representation of data is very similar to a traditional column store layout. Therefore, almost all the compression schemes for column stores presented in [18] can be applied for this layout as well.

F. Discussion

In the *clustering by version* approach, both time travel and evolution of data queries are expected to be expensive for a large number of updates because a lot of tuples have to be scanned. Yet, insert, update and delete operations are simple look-ups and appends and therefore very efficient (constant time).

VII. Hybrid

This section describes a layout representing a hybrid approach with two different types of clustering. The goal of this layout is to limit the amount of data which needs to be scanned to retrieve a given version.

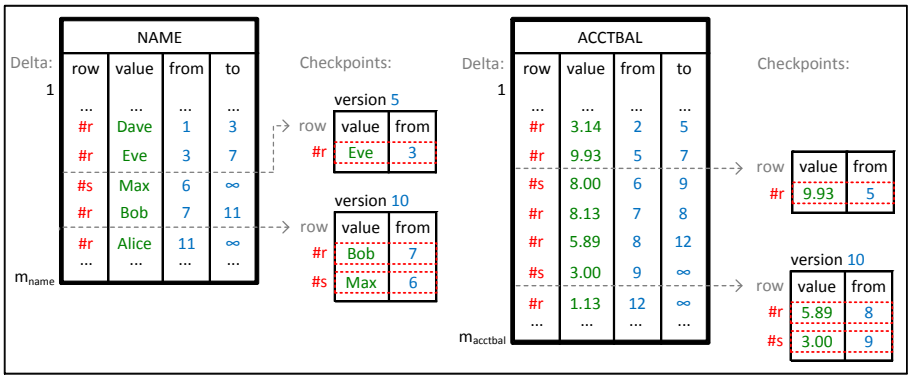


Fig. 4: Hybrid Layout with 2 Checkpoints

A. Storage Layout

The layout of the *hybrid* approach illustrated by Figure 4 is similar to *cluster by version* layout in Section VI, but it includes additional checkpoints, each containing the latest version for all rows at the time that the checkpoint has been computed.

Again, if row with ID i is inserted or updated at version ver, the tuple (i, val, ver, ∞) is appended to a data structure called *delta array* as described in Section VI. The tuples are therefore clustered by version. After a fixed number of updates (defined by the *checkpoints interval* parameter $delta_{max}$) a consistent view of the entire column for the current version is serialized and stored in a *checkpoint*. In such a *checkpoint*, the value val and the latest version ver are stored for each row. The ID of a row is represented implicitly by the position in the *checkpoint*. Hence, a checkpoint is clustered by row.

For keeping track of the versions for which a checkpoint is available, an index is introduced. By means of this information, the last *checkpoint* before a given *version-ID* can be determined efficiently in $O(log(s))$ with s being the number of checkpoints for this column. As the current version is accessed most frequently, a checkpoint of the latest version of each row is always maintained and called *current checkpoint*. When the current version of a row is updated, the old value is removed from the *current checkpoint* and appended to the delta array. The new value is now written to *current checkpoint*.

For this layout, $delta_{max}$ is the most important parameter which defines the maximum number of tuples to be stored before a full checkpoint is computed. A comparison on how to choose this parameter can be found in Appendix C.

B. Query and Update Processing

Insert. In the hybrid approach, the insert operation is a simple append to the *current checkpoint*. If, due to insertion, the size of the table exceeds the maximum size of the *current checkpoint* the data will be copied into an array with double size. For the sake of simplicity, this case is not considered when calculating the memory consumption.

Update. Before the new value is written to the *current segment*, the previous value of the updated row has to be written to the *delta array*. If the number of updates exceeds the limit defined by $delta_{max}$, a new checkpoint is built and a full serialization is written to a newly allocated *checkpoint*. This can be achieved efficiently by a copy operation from *current* to the new *checkpoint*. The position and the *version-ID* that lead to a full serialization are appended to the *checkpoints index* to keep track of the position of all *checkpoints*.

In the *hybrid* layout, similarly to *cluster by version*, both *asynchronous columns* and *synchronous columns* update granularities are feasible. In addition, the synchronization of columns can be achieved based on checkpoints which limits the number of tuples that have to be scanned for retrieving the value in each column.

Delete Operation. The implementation of the delete operation is similar to *clustering by version* approach described in Section VI-B.

Select Value for a Given Version and Row-ID. For retrieving the value for a given row-ID i and version ver, both the clustering by row and by version can be exploited in the hybrid approach. First, the position of the latest previous *checkpoint* is retrieved using the *checkpoints index* (Binary Search $O(log(s))$). The value for the given row-ID can be retrieved from this *checkpoint* by a simple array lookup. Next, the *delta array* is scanned as long as $ver_{from} \leq ver$ to check if the row has been updated. Appendix D provides more details for this operation.

As in the *hybrid* layout the current version is always contained in a checkpoint, this layout provides the fastest possible access to the latest version by means of a simple array lookup.

Select Value for a Given Version and a Group of Rows. Similar to the above mentioned approach, values for a group of rows can be selected within only one single scan.

Aggregation over a Version-Interval for a Single Row. As the aggregation has to be computed for a time interval $[ver_{start}, ver_{stop}]$, the index can be exploited to find the latest checkpoint before ver_{start}. Next, the table has to be scanned

similarly to Section VI as long as the current version is in the time interval.

C. Uncompressed Memory Consumption

In this layout, memory consumption is calculated similar to *clustering by version* approach. In addition, the space for checkpoints and their index has to be considered. The size of each checkpoint can be calculated by:

$$size_{checkpoint} = size(pos) + count_{row} * (size(T) + size(ver))$$

Where pos is the latest position in the *delta array* at time of construction of the checkpoint. The *checkpoint index* is an array constituted of pointers to checkpoints and the versions at which the checkpoints were built. So the index size is:

$$size_{index} = s * (size(ptr) + size(ver))$$

For the sake of simplicity we assume a constant size of all checkpoints. Finally, the total memory consumption is derived by the following formula:

$$size_{total,hybrid} = size_{total,byversion} + s * (size_{checkpoint}) + size_{index}$$

D. Archiving

For storing old versions on harddisk, a checkpoint taken at $ver_{archive}$ is chosen as a threshold. All previous checkpoints and tuples in the *delta array* before that checkpoint are moved on harddisk and deleted from main memory. Again, the checkpoint at $ver_{archive}$ has to be preserved in main-memory to be able to reconstruct the values of all rows. This method prevents the execution of a full table scan operation as it is the case in Section VI.

E. Compression

The *hybrid* layout can be understood as a *cluster by version* approach with additional checkpoints. Therefore, compression of the *cluster by version* layout can be applied as described in Section VI. In addition, checkpoints can be compressed by exploiting the similarity of adjacent checkpoints. We consider a specific number of checkpoints as reference checkpoints and the rest as intermediate ones. Each intermediate checkpoint can now be represented based on the differences compared to its previous reference. Such a representation results in a sparse matrix which can be compressed by means of Yale format [21]. This representation both leads to a reduced memory consumption and a fast reconstruction of checkpoints by only one reference comparison. Again, dictionary encoding [18] and dictionary compression [19] can be applied in addition.

F. Discussion

The advantage of the *hybrid* layout is the speed up for time travel queries for a given row. The execution time of this query is limited and shorter for smaller checkpoint intervals. However, this involves a time-space tradeoff because memory consumption increases for a larger number of checkpoints.

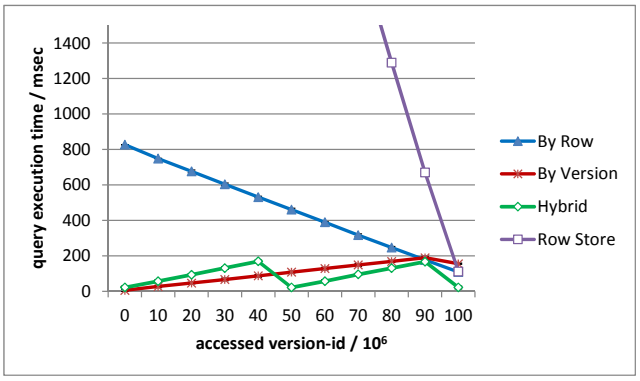


Fig. 5: Time to Select a Given Version from One Column

VIII. Experiments and Results

This section describes the results of performance experiments motivated by use cases from SAP introcuded in Section III involving analytical queries on large data-warehouses containing temporal data. In our experiments, we will study two metrics: memory consumption and the response time of queries and update operations.

A. Experimental Environment

We implemented the memory layouts as a prototype written in C++. The experiments were measured on a Windows Server 2008 64 Bit. The hardware was an IBM x3550 M2 server with 24 GiB RAM and an Intel Xeon L5520 CPU running at 2.26 GHz.

Our benchmark is based on TPC-H with additional update scenarios to generate realistic temporal data. For our measurements we chose the *lineitem* table from the TPC-H benchmark because this table is updated frequently. This table was populated by an initial load of 10 million rows followed by 200 million updates. For each update, the updated attribute was chosen randomly according to a Zipf distribution. In approximately 50% of the updates the value of the *l_quantity* attribute was updated.

The results were compared to an in-memory row-oriented database system as a baseline. Versioning support was implemented in this row store by chaining previous versions. The row store is implemented as an array of tuples each containing the values of different attributes. For each of these tuples a reference to a chain of previous versions is stored. For inserting a new row, a new tuple is appended to the array. Correspondingly, the update operation adds a new tuple to the chain of versions of a row. This idea is similar to [4].

B. Query Response Time Experiments

The queries in this subsection are evaluated on one column only. For the measurements we chose the *l_quantity* from the *lineitem* table.

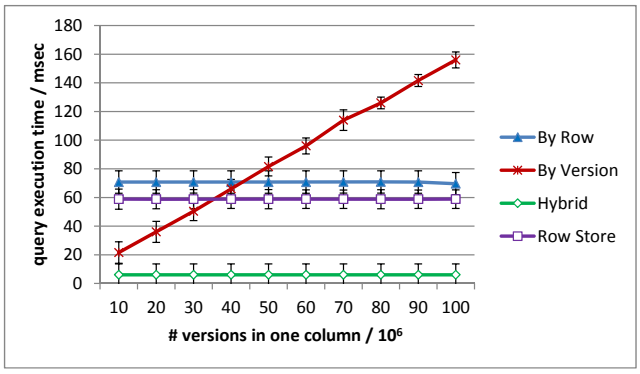


Fig. 6: Time to Select the Latest Version

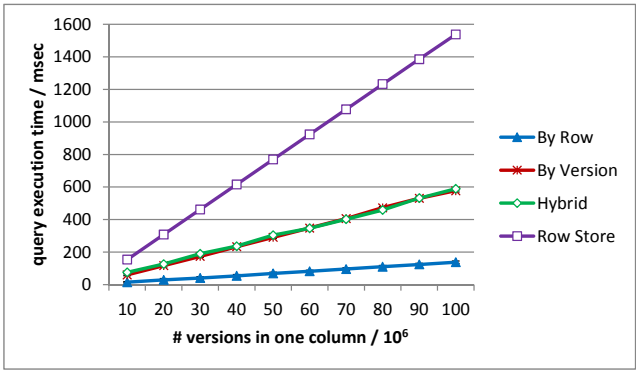


Fig. 7: Time to Aggregate over all Versions of a Row

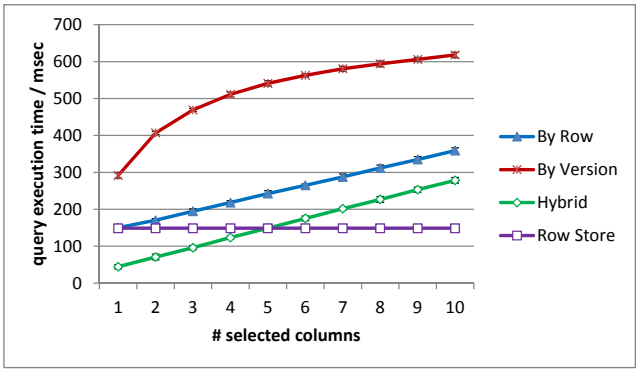


Fig. 8: Select Latest Version from 10 Columns

1) Time Travel to a Previous Version: The measurement results shown in Figure 5 refer to the time travel use case (Section III-A) in which the maximum value of the *l_quantity* attribute for all rows at a given version is calculated. In this diagram the execution time of a query is measured which performs a time travel to a variable previous version.

For the *clustering by row* layout, the query execution time decreases for higher *version-IDs*. This is due to the fact that the newest versions are stored in the leftmost *segment*. The performance decreases linearly for older versions because an increasing number of *segments* in the *overflow array* has to be read.

In contrast to this, the execution time increases for later versions in the *clustering by version* approach because more tuples have to be scanned for a higher *version-ID*.

The performance of the *hybrid* approach decreases faster than the *clustering by version* layout because there is an additional overhead caused by searching for the nearest checkpoint in the index. The sawtooth shape of the line is caused by the execution time increasing linearly with the distance to the nearest checkpoint. For the experiments, only one checkpoint was created. In addition, the latest version can always be retrieved in constant time from the *current* checkpoint.

The performance of the *row store* decreases significantly for lower *version-IDs* because a pointer has to be followed for each version.

2) Select Value for the Latest Version: As a special case of the time travel use case (III-A), Figure 6 shows the execution time of the query that retrieves the latest version of the *l_quantity* attribute for all rows. The query execution time is measured for a variable number of versions which are stored in this column.

For the *clustering by row* layout, the execution time is independent of the number of versions because the latest versions are always stored in the leftmost *segments* and can therefore be accessed directly. In the *clustering by version* layout, the performance decreases steadily with the number of updates because the number of tuples to scan increases. The execution time of accessing the latest version in the *hybrid* layout is constant and better than with *clustering by row* because all data can be read from the *current* checkpoint. Within the checkpoints the accessed values are located closer together which results in a smaller number of jumps in memory and better cache efficiency compared to the wider segments of the *clustering by row* layout. In this experiment, the performance of the row store is similar to *clustering by row* because both approaches store the latest version at the leftmost position.

3) Aggregation over a Version-Interval for a Single Row: According to the audit use case (Section III-B), Figure 7 shows the results for calculating the maximum value of the *l_quantity* column within a version interval for a given row. Again, the execution time of the query is measured for a variable number of versions in this column.

The *clustering by row* layout yields the best result in this case because all versions of a row are clustered together and can be read sequentially. Both for the *clustering by version* and the *hybrid* approach, a full table scan is required to retrieve all versions of a row. This full traversal of all data leads to a worse performance compared to *clustering by row* because an additional comparison with the *row-ID* is required. The query execution time for the row store is the worst because of pointer chasing for each version.

C. Record Reconstruction

Up to now, the performance for executing queries has been shown for a single column only. In this section the record

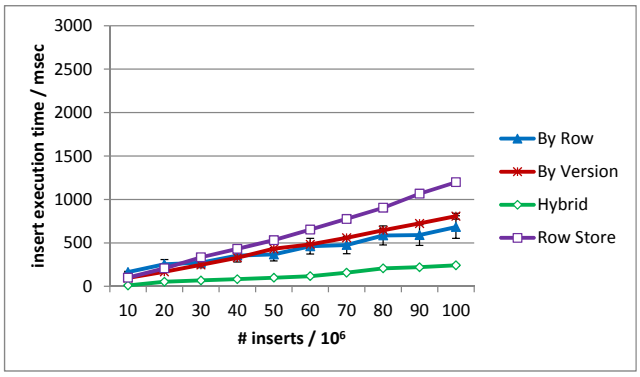


Fig. 9: Insert Execution Time for Variable # Inserts

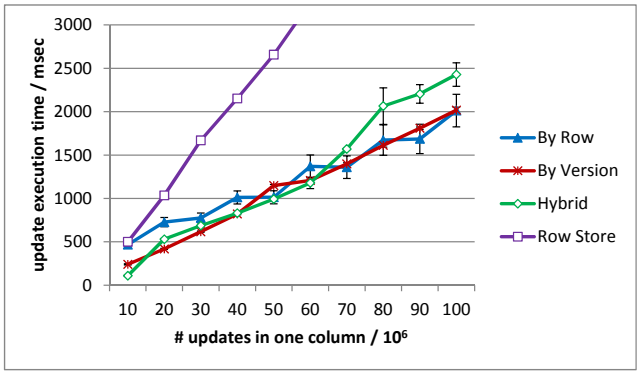


Fig. 10: Update Execution Time for Variable # Updates

reconstruction is investigated by measuring the execution time for different numbers of selected attributes. As an example, Figure 8 shows the execution time of a query which selects the latest value for each row in each column.

For the *hybrid* layouts and *clustering by row*, the latest version is directly accessible from the *current checkpoint* and the leftmost position, respectively. Therefore the performance of the record reconstruction join operator is independent of the number of updates per column which leads to a linear increase with the number of columns. In *clustering by version* for each column a scan has to be performed. Since we are using asynchronous columns, the number of tuples in different columns are not equal. Therefore scanning columns with fewer tuples takes less time and as a result we see slower growth in the right part of the curve. Since in the row store the values for all columns are located in the same record, the execution time does not depend on the number of retrieved columns.

D. Processing Inserts and Updates

1) Inserts: Figure 9 shows the time for inserting a variable number of values into 10 columns of the *lineitem* table. Since inserting a new row is a simple append to the end of array in each layout, we can see that the execution time increases linearly with the number of new rows and the measurement results are in the same order of magnitude for all layouts.

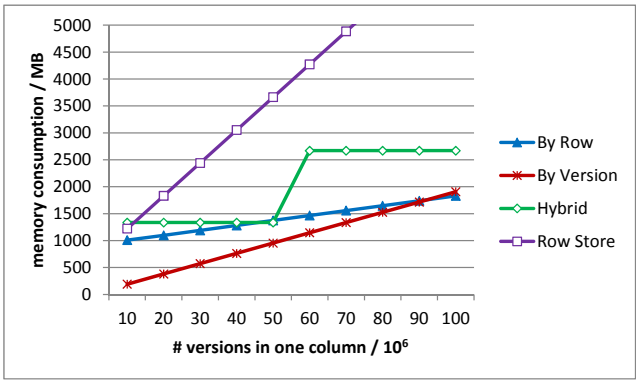


Fig. 11: Memory for one Column with 10 Million Rows

2) Updates:

A value in the database can be updated by inserting a value with a new version for an existing row. Figure 10 visualizes the time for executing a variable number of updates on the *l_quantity* column for the different layouts. The update execution time is approximately in the same order of magnitude for all column store layouts because a new version has to be appended for each approach. In contrast to this, the update performance of the row store is much worse because all attributes of a row have to be replicated for each update.

E. Memory Consumption

Figure 11 visualizes how the memory consumption for the column *l_quantity* scales in all layouts with the number of updates. Compression is disabled for this measurement.

The *clustering by row* layout consumes more memory than the *clustering by version* layout because of the constant width of 10 versions per *segment*. The most memory-efficient layout is *clustering by version* because the amount of unused memory is minimal. The memory-consumption of the *hybrid* approach is higher than *clustering by version* due to existence of checkpoints. The row store is the most memory-inefficient layout because of replicated data.

Figure 12 shows the total memory consumption of a subset of 10 columns from the *lineitem* table. The *clustering by column* approach is the most memory-efficient for 10 columns because all columns are updated independently. The memory consumption of the *hybrid* approach depends on the number of checkpoints. For our experiments, only one checkpoint has

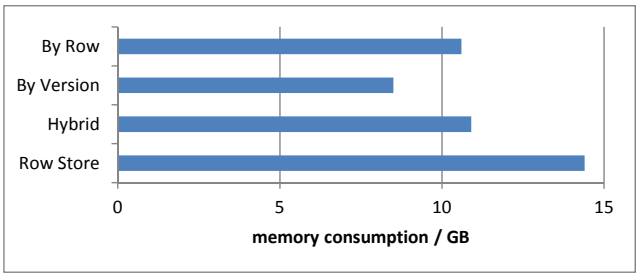


Fig. 12: Memory of 10 Columns with 200 Million Updates

been created. Again, the row store has the highest memory-consumption because it has to replicate the information of all 10 columns even if only a subset of the attributes is updated.

IX. Conclusion

This paper presented three alternative memory-layouts to implement versioning and time travel in column store database systems. The first approach clusters the data by *row-ID* as a natural extension of the current design of column stores. The second layout clusters the data by *version-ID* similarly to the approach taken in log-structured file systems and PostgreSQL. Third, a hybrid approach was presented which is similar to the way that document versioning systems such as RCS and SVN keep versions.

Comprehensive performance experiments studied the trade-offs of all three approaches. As a baseline for the comparisons, an implementation of versioning and time travel in a row store was used. The experiments showed that, overall, all three approaches to implement time travel in a column store outperform the row store, regarding to both query response times and storage overhead. In terms of query response times the *hybrid* approach is the overall winner.

There are a number of interesting avenues for future work. This work was focused on main memory databases because that seems to emerge as the state-of-the-art. Nevertheless, we are planning to repeat the experiments for disk-based and SSD-based database systems. Furthermore, we intend to study more complex workloads that involve indexes. Finally, we are planning to exploit the proposed techniques for an efficient implementation of multi-version concurrency control in column stores.

References

[1] J. Gray, "Transaction research history and challenges, invited talk," *SIGMOD, Chicago, IL*, 26 June 2006.

[2] R. Rajamani, "Oracle total recall / flashback data archive," Oracle, Tech. Rep., 2007.

[3] M. Stonebraker, "The design of the postgres storage system," in *VLDB '87*, 1987.

[4] D. Lomet, R. Barga, M. F. Mokbel, and G. Shegalov, "Transaction time support inside a database engine," in *ICDE '06*, 2006.

[5] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD '08*, 2008.

[6] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz, "Positional update handling in column stores," in *SIGMOD '10*, 2010.

[7] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *SIGMOD Record*, vol. 40, no. 4, pp. 45–51, 2011.

[8] G. Ozsoyoglu and R. Snodgrass, "Temporal and real-time databases: a survey," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 7, no. 4, Aug. 1995.

[9] D. B. Lomet, M. Hong, R. V. Nehme, and R. Zhang, "Transaction time indexing with version compression," *PVLDB*, vol. 1, no. 1, pp. 870–881, 2008.

[10] K. Jernigan, "Oracle total recall with oracle database 11g release 2," Oracle, Tech. Rep., 2009.

[11] A. Held, *Oracle 10g Hochverfgbarkeit. Die ausfallsichere Datenbank mit RAC, Data Guard und Flashback (Edition Oracle)*. Addison-Wesley, 2004.

[12] D. B. Lomet and F. Li, "Improving transaction-time dbms performance and functionality," in *ICDE '09*, 2009.

[13] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Comput. Surv.*, vol. 31, no. 2, 1999.

[14] C. S. Jensen and R. T. Snodgrass, "Temporal data management," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, no. 1, 1999.

[15] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *SIGMOD '85*, 1985.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented dbms," in *VLDB*, 2005, pp. 553–564.

[17] V. S. Inc., "The vertica analytic database technical overview white paper," Vertica, Tech. Rep., 2012.

[18] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.

[19] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *SIGMOD '09*, 2009.

[20] K. Jouini and G. Jomier, "Avoiding version redundancy for high performance reads in temporal databases," in *DaMoN*, 2008, pp. 41–46.

[21] R. P. Tewarson, *Sparse Matrices (Part of the Mathematics in Science & Engineering series)*. Academic Press Inc., 1973.

APPENDIX

A. Clustering by Row

The pseudo code in Algorithm 1 shows the algorithm to access the value for a given ID and version. Note that always the first element in the segment shows the latest version, whereas other versions are stored in increasing order (in order to prevent shifting the array while prepending).

Algorithm 1 Cluster by Row: Get Value for ID and Version

```
function GETVALUE(id, version, base, overflow)
    segment ← base[id]                          ▷ get segment for the given ID
    repeat
        if segment.pair[0].version <= version then
            return segment.pair[0].value
        end if
        for i = colWidth − 1; i >= 1; i − − do
            if segment.pair[i].version <= version then
                return segment.pair[i].value
            end if
        end for
        row ← overflow[row.nextOverflowPosition]
    until row = empty
    return NOT_FOUND
end function
```

B. Clustering by Version

Algorithm 2 shows a scan for the retrieval of a given version and row-ID. The matching value has been found with the occurrence of the first tuple for which the row-ID is equal to the given row-ID and the requested version is in the interval of versions when the value is valid for this row. As a heuristic, a backward scan is performed when the given version is closer to the maximum available version $version_{max}$.

Algorithm 2 Cluster by Version: Get Value for ID and Version

```
function GETVALUE(id, version, data)
    if version < version_max/2 then
        for i = 0; i < data.size; i + + do
            if data.from[i] <= version <= data.to[i] then
                if data.id[i] = id then
                    return data.val[i]
                end if
            end if
        end for
    else
        for i = data.size; i > 0; i − − do
            if data.from[i] <= version <= data.to[i] then
                if data.id[i] = id then
                    return data.val[i]
                end if
            end if
        end for
    end if
    return NOT_FOUND
end function
```

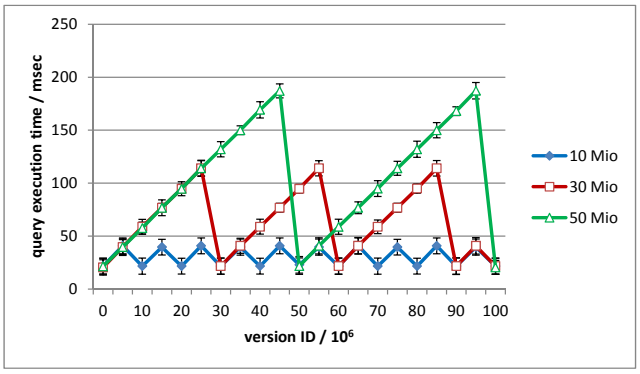


Fig. 13: Serialization Interval for the Hybrid Approach

C. Serialization Interval for the Hybrid Approach

In the *hybrid* layout described in Section VII, the *checkpoints interval* $delta_{max}$ defines the maximum number of updates that are stored in the *delta array* before a *checkpoint* is generated. This involves a time-space tradeoff. Figure 13 shows the query performance for accessing a *version-id* for three different *checkpoints intervals* in one column. A smaller *checkpoints interval* leads to a smaller amount of data in the *delta* to be scanned and a smaller maximum execution time on one hand, but causes increased memory consumption due to larger number of checkpoints on the other hand. For our experiments with the *hybrid* approach, we chose 50 million as the checkpoints interval ($delta_{max} = 50\ Million$), meaning that after every 50 million updates we build a new checkpoint.

D. Hybrid

In algorithm 3, first a pointer to the nearest checkpoint older or equal to the given version is retrieved by applying a binary search in the *index*. This search retrieves the position of the largest checkpoint smaller or equal to the given version. If the given version is equal to the version at which the checkpoint was built, the value for the *row* can be obtained in a single array-lookup. Otherwise the delta has to be read in addition until the version at the current position becomes larger than the given version.

Algorithm 3 Hybrid: Get Value for ID and Version

```
function GETVALUE(id, version, delta, checkpoints, index)
    indexPos ← binarySearch(index.versions, version)
    checkptOffset ← index.checkptOffsets[indexPos]
    deltaPosition ← index.deltaPositions[indexPos]
    value ← checkpoints[checkptOffset + id]
    i ← deltaPosition      ▷ read delta as long as version is valid
    while i < delta.size and delta.ver[i] <= version do
        if delta.id[i] = id then
            value ← delta.val[i]
        end if
        i ← i + 1
    end while
    return value
end function
```