Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA

Martin Kaufmann[†§], Amin A. Manjili[†§], Panagiotis Vagenas[†§], Peter M. Fischer[#], Donald Kossmann[†], Franz Färber[§], Norman May[§]

[†]Systems Group
ETH Zürich, Switzerland
{martinka,amamin,pvagenas, donaldk}@ethz.ch

[#]Albert-Ludwigs-Universität
Freiburg, Germany
peter.fischer@cs.uni-freiburg.de

[§]SAP AG
Walldorf, Germany
{franz.faerber,norman.may}@sap.com

ABSTRACT

Managing temporal data is becoming increasingly important for many applications. Several database systems already support the time dimension, but provide only few temporal operators, which also often exhibit poor performance characteristics. On the academic side, a large number of algorithms and data structures have been proposed, but they often address a subset of these temporal operators only. In this paper, we develop the *Timeline Index* as a novel, unified data structure that efficiently supports temporal operators such as temporal aggregation, time travel, and temporal joins. As the *Timeline Index* is independent of the physical order of the data, it provides flexibility in physical design; e.g., it supports any kind of compression scheme, which is crucial for main memory column stores. Our experiments show that the *Timeline Index* has predictable performance and beats state-of-the-art approaches significantly, sometimes by orders of magnitude.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Dictionaries, Indexing methods* ; H.2.4 [**Database Management**]: Systems—*Query Processing*

Keywords

Temporal Data, Temporal Operator, Index, Algorithm

1. INTRODUCTION

Temporal data management is a critical feature in most database systems today. Instead of "update-in-place", modern database systems create a new version of an object. Once the cost of keeping these additional versions has been paid, users expect rich capabilities to query and process that data. For instance, users wish to compare the current status of their investment "portfolio" with the status *AS OF* a year ago. Querying a historical version of the database is typically referred to as *time travel* [25]. Another example is the analysis of how many orders are delayed as a function of time in a quality assurance system, thereby querying all historical versions of the database over a certain time period. This particular application is called *temporal aggregation* [12]. Applications such as Facebook Timeline have brought temporal data and such temporal query operators to the limelight.

From an academic point of view, temporal data management has been the subject of extensive research. Since Snodgrass' seminal work on defining the temporal data model [22], there has been a large body of work in this area, summarized e.g., in [9, 20]. That work covers proposals for index structures (e.g., multi-version B-trees [3]) and algorithms for certain kinds of queries (e.g., temporal aggregation [4, 12] and temporal joins [9, 27]).

From an industrial perspective, the adoption of temporal database technology has been much slower. Based on Snodgrass' proposal of almost twenty years ago [22], SQL has included temporal features only recently as part of the SQL:2011 standard [14]. Even that standard, however, lacks many important features such as temporal aggregation or temporal joins. Database vendors have also been rather hesitant to ship products with temporal features. IBM DB2, for instance, has added support for bi-temporal data only with its latest version, which was released in 2012 [21]. The only exception is Oracle, which has been supporting time travel using its Flashback technology for more than 10 years [18]. But even Oracle Flashback does not support temporal aggregates and joins.

Market traction and lack of incentives is clearly not the problem: Customers are desperate to get rich temporal features. At SAP, for instance, application developers of the financial (FI) and sales & distribution (SD) modules implement temporal operators as part of the application logic because the relational database products do not support these features. Temporal operators are needed for these applications for legal, compliance / auditing, and reporting use cases (e.g., risk assessment). Implementing database functionality in the application is not only bad from a developer's productivity perspective, it also kills performance as large volumes of data need to be shipped from the database server to the application server.

The lack of support for temporal features in state-of-the-art database systems has actually technical reasons. Looking closely at the literature, it turns out that most of the work on temporal data management is highly specialized and proposes index structures and algorithms for a specific temporal function (e.g., temporal aggregation). While all of these functions are important and deserve special attention and tuning, even a global player like SAP cannot afford to implement a new data structure for each kind of temporal query. From a customer perspective, the operational cost of maintaining dedicated index structures on the same data for each kind of temporal query can also be prohibitive.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

To our biggest surprise, the most significant knock-out criterion for the majority of the existing proposals from the research literature was performance. We did extensive experiments with the best-of-breed approaches from the literature and found out that the performance results were simply not acceptable for SAP HANA. Digging deeper, it turns out that many of these proposals do not parallelize well and do not work efficiently on modern hardware with many cores, large main memories, and non-uniform memory access (NUMA). For instance, all approaches that are based on tree structures (e.g., B-trees) showed poor performance in our experiments because, even in main memory, a sequential access pattern is essential in order to avoid contention in the memory system. Another problem with such tree-based structures is that they only work well for queries with high selectivity; i.e., queries that select a few tuples based on either a temporal or spatial criterion. As many of our customer use cases involve analysis over large volumes of data, including significant parts of the temporal data, no approach presented in literature before was applicable for SAP HANA. (Section 6 presents our most important findings and performance results.)

The purpose of this paper is to share our experience with existing techniques to process temporal queries and describe the approach chosen for SAP HANA. The main contribution is to present a novel index structure called *Timeline Index* and the algorithms used to process different kinds of temporal queries on this index structure. In a nutshell, we chose *Timeline Indexes* for the following reasons:

- *Generality:* The *Timeline Index* is a single data structure that can be used to process a large variety of different temporal queries. In particular, we can address all our customer use cases. Besides, only a single *Timeline Index* per table is needed.
- *Performance:* As shown in Section 6, the *Timeline Index* outperforms the best known approaches for each kind of temporal query in our context (i.e., main memory column stores). In some cases, the *Timeline Index* beats the best known existing algorithms by orders of magnitude.
- *Memory efficiency:* The *Timeline Index* is space-efficient. There are space/time tradeoffs, but even in a space-consuming variant, the storage overhead is roughly only 15 percent.
- *Flexibility:* Many traditional techniques require that tables are ordered by *system time*. This requirement limits the physical design and can result, among others, in poor compression. The *Timeline Index* sheds this limitation and can be used independently of other decisions for physical database design.
- *Applicability:* The *Timeline Index* can be integrated naturally into the HANA system. It can be implemented as a normal table, thereby reusing the same structures and algorithms that are already in place to efficiently maintain and process tables.

The main ideas of the *Timeline Index* and the algorithms presented in this paper are general: In principle, they can be applied to both row and column stores. Nevertheless, the focus of our work has been on main memory column stores because a core part of HANA is exactly that kind of system. One may argue that temporal databases grow so large that it is not economic to keep all information in main memory. Still, utilizing compression, as well as the continuing trend of larger main memories, and distribution over clusters of machines, HANA is already able to handle (temporal) queries on hundreds of terabytes of data in main memory.

The remainder of this paper is organized as follows. Section 2 gives an overview of existing work on temporal data management. Section 3 briefly describes HANA, our target database system. Section 4 presents the *Timeline Index*. Section 5 introduces algorithms on how to process different kinds of temporal operators using the *Timeline Index*. Section 6 gives the results of a comprehensive performance study that compares the *Timeline Index* with existing approaches for a variety of temporal queries. Section 7 contains conclusions and possible avenues for future research.

2. STATE OF THE ART

Following Snodgrass' work on defining the foundations of the bi-temporal data model in the early 1990s and the resulting TSQL2 standards proposal [22], a large body of research on temporal data has been established. Various algorithms and data structures have been proposed for different temporal operators. We will provide a brief overview, covering the parts which are relevant for our general design and our specific use cases.

2.1 General Temporal Indexes

A first important direction of work is given by general methods to model and organize temporal data. A survey by Salzberg et al. [20] lists the typical access patterns (*Timeslice*, *Key in Time* and *Key/Time range*) and provides an overview of how well various index structures support these operations. Since most of these index structures were developed in the mid-to-late '90s, they are designed for hard-disk efficiency, optimizing the number of I/O operations for updates and queries. Tree indexes over intervals or versions are used, relying on various clustering strategies for time and key values, and partial replication for efficiency. Furthermore, some index types were designed to "truncate" history with the goal of moving it to other storage media ([15]), but distribution and parallelization have not been researched widely. Given the design goals, some proposals (such as [3]) have been proven to have (asymptotically) optimal I/O behavior for a range of temporal queries. However, given the different tradeoffs between access time, transfer speeds and CPU cost, these structures will not necessarily perform best in a main-memory setting.

We will briefly discuss those indexes that are most relevant to us:

The Time Index [6] (from now on referred to as *Elmasri 1990* for clarity) is one of the earliest temporal indexing methods, and provides explicit support for all our use cases. It is directly comparable to our proposed *Timeline Index* because it indexes only the time dimension. Technically, the Time Index is a B^+-Tree over versions, in which each leaf page contains all active versions at the beginning, and the changes afterwards. The multi-version B-tree [3] (mentioned as *MVBT*) is one of the most advanced temporal indexing methods. It provides an index for both key- and time-dimensions with optimal I/O behavior. As a result, it is able to support many query classes and exploit clustering over the time and key space. Its implementation is based on a (logical) forest of B-Trees sharing pages. In contrast, [19] provides a much simpler index structure, based on a single B^+-Tree and encoding of windows over intervals, making it the closest match to the Timeline Index. The query performance of [19] is also optimal; the complexity of updates has not been studied yet.

2.2 Temporal Aggregation

A challenging temporal operator is temporal aggregation, in particular *temporal grouping*. In contrast to non-temporal, traditional aggregates, temporal grouping computes the aggregates as running values for time points or time intervals; e.g., the number of sales that occurred for each point in time. Temporal grouping and aggregation are well-researched topics. Snodgrass et al. [12] introduced the first algorithm for computing temporal aggregation on constant intervals. In this algorithm, for each aggregation function and each attribute a separate data structure called *Aggregation Tree* is built.

Since this tree is not guaranteed to be balanced, it may degrade into a linked list. In order to overcome the worst case, several variants of the Aggregation Tree have been proposed. However, these variants usually make special assumptions about the distribution of intervals or suffer from the drawbacks of the original design.

Böhlen et al. [4] introduced an algorithm for temporal aggregation based on AVL Trees for *start* and *end point* values. The temporal aggregation is performed by traversing the *start* index and inserting the tuples that are activated into the *End Point Tree*. The tuples which expire are removed from the *End Point Tree* (tuples are removed in their *end* time order), and the aggregate is returned as a result. The actual cost depends on the type aggregate: While cumulative aggregates like `SUM` and `COUNT` require little storage and cost, certain aggregates such as `MAX` drive up the resource requirements. We will discuss these issues in more detail in Section 5.1 because they are relevant for the design of how to process temporal aggregates with the *Timeline Index*, too.

Some of the general-purpose temporal index structures (e.g., [6]) are useful to compute temporal aggregates. Nevertheless, it is worth noticing that some of them, as for instance the MVBT tree, do not support temporal grouping and are limited to certain aggregates like `SUM` and `COUNT` (e.g, the MVSB tree [28] which is based on the SB-Tree [26]).

2.3 Time Travel

Establishing a consistent view of a (past) version of a database is currently the most widespread use case of temporal operations. Several database management systems provide support for this operation, which is typically called *time travel*.

Oracle pioneered time travel with its Flashback feature [18], which is integrated into the Oracle database product. IBM DB2 also provides support for management of temporal data and time travel [21]. PostgreSQL offers a similar feature based on the append-only design of the PostgreSQL storage manager [25].

SAP HANA [7] (which is described in more detail in Section 3) provides a basic form of time travel queries based on restoring a snapshot of a past transaction. ImmortalDB [16] by Microsoft Research is another system that supports versioning and time travel queries by chaining versions of records and navigating to the appropriate version of a record. The indexing data structure used in ImmortalDB is a TSB-Tree [15] which defines a time range for each page in memory and keeps the data for the versions related to this time range in the corresponding page. It is therefore expected that the time travel operator performs quite well by accessing exactly the pages that contain the data related to the target version.

Furthermore, Time Travel is supported by general-purpose temporal index structures such as *Elmasri 1990* [6] and MVBT [3].

2.4 Temporal Join

Temporal Joins contain predicates on both key and time domains. Typically, two tuples are considered to be join candidates on the temporal domain if their version ranges overlap.

There are two classes of algorithms for temporal joins: 1) Index-based algorithms that use extra data structures for identifying tuples or their locations, either based on their join-attribute or on their temporal properties. 2) Non-index algorithms that directly work on the temporal tables. A comprehensive survey on existing join algorithms with support for temporal tables is provided by [9]. According to this survey, a valid-time natural join [23] can be evaluated in three different ways [9]: Using nested-loop-based, sort-merge-based and partition-based algorithms.

Elmasri 1990 [6] supports temporal joins by building a two-level index which combines a B^+-Tree index over the join attribute with a B^+-Tree index over the time dimension. The idea is that each leaf node of the top-level index (B^+-Tree) includes a value of the search attribute and a pointer to a separate index. Hence, there is an index for each attribute value. This method suffers from high memory consumption. As an alternative, [27] proposes several join algorithms which exploit MVBTs, for instance clustering in space and time, replication of records and linkage.

Method	Time Travel	Temp. Aggr.	Temp. Join
SB Tree	no	yes ([26])	no
Böhlen 2006	no	yes([4])	no
TSB-Tree	yes ([15])	no	no
Elmasri 1990	yes ([6])	yes ([6])	yes ([6])
MVBT	yes ([3])	(MVSB) ([28])	yes ([27])
Timeline Index	yes	yes	yes

Table 1: Supported Operations for different methods

In summary, we can make three observations that motivated our work on a new, uniform and general-purpose data structure to support temporal queries: First, several general-purpose temporal index structures exist, but they are not tuned for large main memories and modern hardware. Second, production systems only support one particular kind of temporal query (i.e., time travel), even though there is demand for all kinds of operators. Third, there is significant work on the other two types of queries (i.e., temporal joins and temporal aggregation), but all these approaches have shortcomings which limit adoption in production systems. Table 1 summarizes these findings on operators and specific access methods.

3. THE SAP HANA DATABASE SYSTEM

SAP HANA [7] is a commercial database system which employs both a column store and a row store for in-memory data processing.

3.1 Architecture of SAP HANA

SAP HANA was designed for supporting modern hardware such as multi-core systems and large main memories. Especially fast full column scans and customized operators as well as massive intra- and inter-operator parallelism contribute to the performance characteristics. Column stores are well suited for analytic queries on big amounts of data, which originally was the core business of HANA. Currently, HANA is being extended to be able to handle both OLAP and OLTP workloads efficiently in one system.

HANA makes use of multiple compression schemes to reduce the main memory consumption and improve query execution times. To achieve high insert/update performance, updates and inserts are first applied to one or multiple non-compressed *delta stores* which are specifically tuned for high volumes of updates. HANA periodically merges the new data from the delta stores into the *main store* which is tuned for efficient reads. In order to guarantee consistency, all operations (in particular queries) take deltas and main stores into account.

HANA is a distributed database system which allows the deployment of multiple servers for a single database. The biggest installation (in our lab) so far consists of 250 nodes with 1 TB each, which sums up to 250 TB of main memory. With an average compression ratio of 5, this installation can load up to 1.25 PB of raw data. HANA includes multiple engine types such as a text engine, a graph engine, an OLTP engine, and others. Concurrency control is implemented in the OLTP engine using Snapshot Isolation, which is an important prerequisite to implement clear semantics for temporal data management.

Basic support for temporal data (transaction time) is already available natively in HANA, but only a subset of possible op-

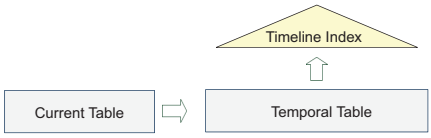


Figure 1: Architecture Overview

erators are currently supported. The most prominent examples for temporal data structures are the data store objects in the SAP Business Warehouse product (BW, DSO) and the so-called "change documents" in the SAP ERP system, where applications store the history of business objects.

3.2 Integrating Timeline Indexes into HANA

The *Timeline Index* is implemented as a prototype based on the architecture of HANA. We designed the data structures and algorithms to fit the properties of modern hardware and with the goal to be implemented into the SAP HANA product. Given this perspective of a real system integration, all aspects of productive software like performance, memory consumption, parallelism, and complexity of algorithms had to be taken into account. Therefore, the data structures should be simple, the memory overhead must be low, incremental updates have to be supported, delta structures as well as fast index reconstruction must be available. The *Timeline Index* is general and can be applied to both the column store and the row store of HANA. As temporal queries are often part of OLAP workloads, however, we envision that *Timeline Indexes* are mostly used with a columnar table layout.

4. TIMELINE INDEX

This section describes the data structures and basic principles of the *Timeline Index*. Based on these data structures, Section 5 describes the algorithms used to implement various kinds of temporal operators.

4.1 Fundamentals and Overall Architecture

The lower part of Figure 1 shows how HANA manages temporal data [7]. The same architecture has been adopted by DB2 [21]. For every table, HANA keeps the *current* version of the table and the whole history of previous versions of the table in separate structures. For simplification we assume in this paper, that the *current* version is always replicated to the *Temporal Table*. The *Current Table* provides efficient access to the current state of the database as such accesses are the most common use cases for HANA. Temporal features (e.g., time travel) are implemented using the *Temporal Table*, and this is where the *Timeline Index* takes effect: It is an index that accelerates operations carried out on a *Temporal Table*. For each *Temporal Table*, there is exactly one *Timeline Index*. *Temporal Tables* and *Timeline Indexes* are the focus of this work.

Our work is based on the standard formalism for bi-temporal data, established by Snodgrass [22]: Each tuple of the *Temporal Table* carries two time intervals $[start_t, end_t)$ and $[start_v, end_v)$, representing *transaction time* and *valid time* (a.k.a. *system time* and *application time*, respectively). For the purpose of this paper, we will focus on the *system time* interval and call this interval $[start, end)$. The timestamps used in these intervals are discrete, monotonically increasing and scoped at the level of a database. In abstract terms, we call these values *Version_IDs*, in the concrete implementation of our system we use *Commit_IDs* of transactions as versions. Since HANA uses Snapshot Isolation for concurrency

Name	Balance
Carl	$100
Ellen	$700

(a) Current Table

ROW_ID	Name	Balance	Start	End
1	Alice	$200	101	103
2	Ann	$300	102	107
3	Carl	$100	103	∞
4	Alice	$500	103	106
5	Ellen	$700	105	∞
6	John	$400	105	106

(b) Temporal Table

Figure 2: Example Current and Temporal Tables

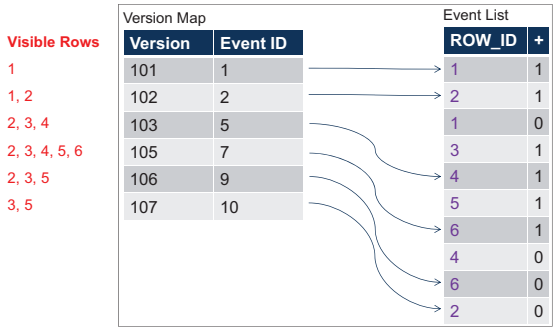


Figure 3: Timeline Index for Temporal Table of Figure 2b

control, these *Commit_IDs* provide discrete and monotonic temporal semantics.

Figure 2 gives an example of a *Current Table* (Figure 2a) and a *Temporal Table* (Figure 2b) in the HANA temporal data model. This example models a small banking application with customer names and their account balance. The name of a customer is assumed to be a key. For brevity, Figure 2 represents the tables as they would be implemented in a row store; however, both *Current* and *Temporal Tables* could just as well be implemented in the HANA column store. Figure 2 shows that the new customer Alice was inserted by Transaction 101. Transaction 102 created customer Ann and Transaction 103 created Carl. In addition, Transaction 103 updated the balance of Alice, thereby invalidating the first version of the Alice record (identified by *ROW_ID* 1) and creating a new version of the Alice record (identified by *ROW_ID* 4). Transaction 104 did not update this table. Transaction 105 created two new customers (Ellen and John) and Transactions 106 and 107 deleted the accounts of Ann, Alice, and John. Figure 2a) shows the current state of the table after all these transactions have been applied; Figure 2b) captures the whole history as needed for temporal queries such as asking when Alice did have more money than Carl.

The *Temporal Table* of Figure 2b) is clustered by the *Start* column. Sorting the table by *Start* sounds like a good idea for temporal query processing and indeed many temporal index structures assume such a design. Unfortunately, this design is not good for compression: Systems like HANA automatically detect the best way to sort a table, optimizing for compression ratio. It turns out that *Start* rarely is the best clustering criterion for the *Temporal Tables* of HANA. Therefore, the *Timeline Index* does not rely on any physical order of the temporal data in memory.

4.2 Timeline Index Data Structure

Figure 3 shows the *Timeline Index* for the *Temporal Table* of Figure 2b. The idea of the *Timeline Index* is to keep track of all the *visible* rows of the *Temporal Table* at every point in time. To this end, the Timeline Index returns all rows that are *activated* or *invalidated* at each point in time. For instance, Row 1 of

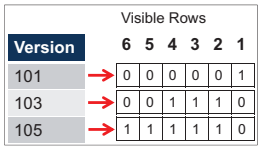


Figure 4: Checkpoint Index

the *Temporal Table* of Figure 2b) is activated at Version 101 and invalidated at Version 103 of the database. The basic idea of the *Timeline Index* has similarities to the LHAM approach [17]; the algorithms are based on Counting Sort [13].

More concretely, a *Timeline Index* consists of two data structures which are scanned concurrently to implement any kind of temporal operation (Section 5). The first data structure is the *Event List*. The *Event List* keeps track of each *invalidation* and *activation* event. Activation events are marked with a "1" and invalidation events are marked with a "0". For instance, the first entry of the *Event List* indicates the activation of Row 1. The second event indicates the activation of Row 2, and so on. The events in the *Event List* must be sorted by the (system) time when the event occurred; i.e., Row 1 was activated before Row 2. The order of events created by the same transaction is undefined; for instance, the order of the invalidation of Row 1 and activation of Row 3 is irrelevant because these events were created by Transaction 103.

The second data structure of the *Timeline Index* is the *Version Map*. The *Version Map* keeps track of the sequence of events that are *seen* by each version of the database; i.e., by each commit of a transaction. This is achieved by storing the *end* offset for each version in the Event ID column. For instance, the *Version Map* of Figure 3 indicates that Version 101 of the database sees only the first event of the *Event List*; Version 103 of the database sees the first five events of the *Event List*; its changes are contained in the range after the second event (last event of the previous version) to the fifth. By concurrently scanning and merging the *Version Map* and *Event List*, it is possible to reconstruct all the visible rows of the *Temporal Table*. All algorithms for temporal operators presented in Section 5 exploit this feature. Figure 3 shows the visible rows for each version of the database in red. Again, this information is implicit and generated while using the *Timeline Index*: It is not materialized because the space overhead would be prohibitive.

Both the *Version Map* and *Event List* can be implemented efficiently using the existing structures of a column store like HANA; i.e., these two structures are implemented as regular tables in HANA and can be scanned and processed just like any other HANA table. The only difference is that these two data structures are *append-only*; that is, once an entry has been inserted into either the *Event List* or *Version Map*, none of its fields will ever be updated. This restriction is acceptable for indexing *system time*, but not for *application time*. We will develop an index for *application time* as part of future work.

Again, it should be noted that only one *Timeline Index* is needed per *Temporal Table* and that the *Timeline Index* is significantly smaller than the *Temporal Table*, in particular, if the table has many columns. Our experiments (reported in Section 6) indicate that a *Timeline Index* is typically only a small fraction of the size of a *Temporal Table*, even if we include the additional space required for checkpoints, which are discussed in the next section.

4.3 Checkpoints

Encoding the deltas between different versions in the *Timeline Index* leads to a compact representation of how data evolves in time, supporting temporal aggregations well. However, reconstructing all tuples which are visible at a given version still requires the traversal of the index up to that version, leading to linearly increasing cost to access (later) versions. In addition, removing old versions for archiving or garbage collection is not possible. To overcome this problem, we augment the difference-based *Timeline Index* with a number of complete version representations at particular points in the history. We call such a full view a *checkpoint*. As shown in Figure 4, a checkpoint is a bit vector which represents the visible rows of the *Temporal Table* at a certain version. In this straightforward implementation, the length of this bit vector is equal to the number of tuples stored in the *Temporal Table* at the time the checkpoint was created. We index these checkpoints by mapping the *Version_ID* at which the checkpoint was taken to the checkpoint contents. In addition, together with the checkpoint, we store the position of the entry in the *Version Map*, so that we can start our scan there.

The cost for accessing a checkpoint is determined by the checkpoint creation policy: If checkpoints are created at fixed version intervals, the location of the latest checkpoint before a given version can be computed in $\mathcal{O}(1)$ by a simple modulo operation. In the example of Figure 4, checkpoints are taken regularly after 2 versions. If instead the distances are more varied (e.g., after a fixed number of operations for the specific table), we have to search, e.g., by using a binary search algorithm. As in practice a relatively small number of checkpoints is needed, even the overhead incurred by a tree search becomes almost a small constant. With this basic implementation we already reach a good tradeoff between storage space, update cost and query performance.

Further improvements are possible by using techniques such as delta checkpoints (storing the difference to a previous checkpoint, trading some computation time for space gains), bit vector compression such as run-length encoding or the Chord [24] bit vector format. Beside the direct benefit for query processing, checkpoints are also aiding archiving old temporal data on disk, garbage collection, parallelization and distribution to different nodes of a cluster by providing clear "cuts". Therefore, we can freely discard versions before the checkpoint, move them to a different location (disk or remote storage) or query the archived data in isolation. This enables storing all temporal data in main memory even if it exceeds the capacity of a single machine.

4.4 Timeline Index Construction

Based on the design of our index, we can now describe how to efficiently create and incrementally update it, even when the underlying data is not in *start* time order. We will first show the bulk algorithm for ordered data and then generalize it. The maintenance algorithms are based on Counting Sort [13], as the index was designed to work with this approach in mind: In a first pass, we count the changed tuples per version and, based on this information, we can create compact *Version Map* and *Event List* tables and fill in the actual *ROW_IDs* in a second pass. The algorithm requires an intermediate table with size equal to maximum *Version_ID*, counting the number of events per version. First, all counters are initialized with 0. Next, at the first linear scan of the *Temporal Table*, we read the *start* time of each tuple, take this value as position in the intermediate table and increase the counter value at this position by 1. In the same pass, we do the same for the *end* time if its value is not infinity. We can now scan the intermediate table, sum up the number of events occurring before the current version and write the value to the *Event ID* column of the *Version Map*, easily determining the offset of the events seen so far. Knowing the total number of versioned tuples from the last *Event ID*, we allocate space for the *Event List*. Now, in a

second linear scan of *Temporal Table*, we write the *ROW_ID* for each *start* and *end* at the *Event ID* given by the *Version Map* and increase this position by one. This results in the events for each version being sorted by *ROW_ID*, which minimizes random I/O. As outlined above, we add a "true" to the bit vector if the tuple is activated at this version and a "false" if it is invalidated.

The overall cost of this algorithm is linear with respect to the size of the *Temporal Table* since it needs to touch each tuple only twice – once for counting the number of events per *Version_ID* and once for writing the values to the *Event List*. The physical order of the data is irrelevant, since the Counting Sort of all *Version_IDs* is performed by the intermediate table.

Furthermore, the index can be updated incrementally by just appending the new versions and the corresponding events to the *Timeline Index* if the *Temporal Table* is sorted by *start* time. Storing the *Temporal Table* in a different sort order, e.g., for achieving a better compression, incurs additional effort which is dominated by resorting the *Temporal Table*. In the latter case, the *ROW_IDs* are not stable any more and need to be updated in the *Timeline Index*, which can be done in linear cost. Alternatively, the index could be dropped and recomputed.

Finally, in contrast to algorithms on other temporal data structures ([1, 3, 6]), this scan-based algorithm expressing version differences lends itself well to parallelization and distribution.

5. HISTORY OPERATORS

The *Timeline Index* has been designed to provide efficient support for a wide range of temporal queries. This section covers three common types of temporal queries and shows how the *Timeline Index* supports processing these queries: (a) Temporal Aggregation, (b) Time Travel, and (c) Temporal Joins.

5.1 Temporal Aggregation

The first temporal operator we discuss is temporal aggregation. A typical example is a query that asks for the most expensive product at each point in time. First defined in [12], temporal aggregation involves the execution of an aggregate function for each *Version_ID*; i.e., each state in which the database has ever been in. Implementing temporal aggregation is demanding because it requires aggregation along the time *and* the spatial dimensions (e.g., *product*) and both of these dimensions have potentially many values. Being so challenging, temporal aggregation has already attracted considerable attention in the research literature [12, 4, 28]. The complexity of the temporal aggregation operator depends on the kind of aggregate: selective aggregates such as `MIN`, `MAX`, and `MEDIAN` are more complex than cumulative aggregates such as `SUM`. Therefore, we describe these two kinds of aggregates separately in the following subsections.

As already mentioned, temporal aggregation has not yet been standardized as part of SQL. For the purpose of this paper, we express it with a special `GROUP BY VERSION_ID()` clause [11].

5.1.1 SUM, AVG, and COUNT

Example. What is the total sum of the account balances of all customers whose name start with "A" at each point in time? This query can be expressed as follows:

```
SELECT SUM(balance) AS sum
FROM Customer co
WHERE co.name LIKE 'A%'
GROUP BY co.VERSION_ID();
```

`SUM`, `AVG`, and `COUNT` are cumulative aggregates which means that a new aggregate value can be computed directly from the previous aggregate value and the changes between the next and previous version of the database. As a result, this kind of aggregation functions is the simplest case for temporal aggregation. Figure 5 shows how such temporal aggregates can be computed using a *Timeline Index*. For the sake of illustration simplicity, we use a shortened representation of the *Timeline Index*, which only lists the *ROW_IDs* for each version and indicates an activation by a "+" and an invalidation by a "−". To compute a temporal `SUM`, we scan the *Timeline Index* to determine the new and invalidated *Customer* rows for each version. Furthermore, we keep a single variable, *sum*, that keeps track of the aggregate value during the scan for each point in time. For each entry of the *Timeline Index*, we check the `WHERE` clause (if the query has one) for all the new and invalidated *Customers*. If a *Customer* qualifies, we look up the *Customer*'s *balance* from the *Temporal Table* and adjust the *sum* variable accordingly (add the *balance* for a new *Customer*; subtract the *balance* for an invalidated *Customer*). This way, the *sum* variable reflects the correct aggregate value for each point time during the scan through the *Timeline Index*.

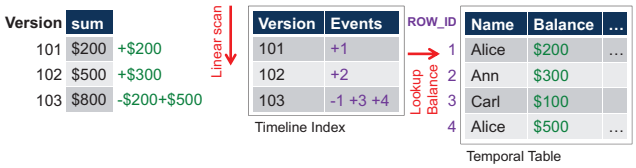


Figure 5: Temporal Aggregation: SUM

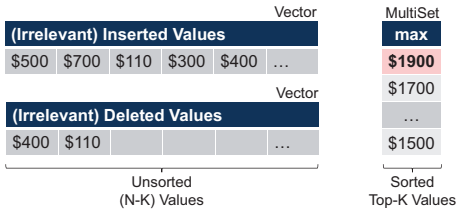


Figure 6: Temporal Aggregation: MAX

`COUNT` aggregates are computed analogously. For `COUNT`, we do not need to look up the *balance* values from the *Temporal Table*; only the `WHERE` clause needs to be evaluated and the running variable that keeps track of the count needs to be maintained. `AVG` is computed from `SUM` and `COUNT`. Likewise, `VARIANCE` and `STDEV` (standard deviation) can be computed from other aggregates.

Complexity. Let N be the number of rows in a temporal table. Let M be the number of events in the *Event List* in the *Timeline Index*. $N \leq M \leq 2 * N$ because each line in the table contributes at most twice to the *Events List* (once for activation and zero or once for invalidation). Since each event is processed exactly once and updates of the aggregation variable have a constant cost, the complexity of `SUM` and `COUNT` is $\mathcal{O}(M)$, which is in $\mathcal{O}(N)$.

5.1.2 MIN, MAX, MEDIAN

Example. What is the price of the most expensive unshipped item at each point in time?

```
SELECT MAX(li.l_extendedprice) AS max_price
FROM Lineitem li
WHERE li.l_linestatus = 'O'
GROUP BY li.VERSION_ID();
```

`MIN`, `MAX`, and `MEDIAN` are selective aggregate functions. That is, we cannot compute the new aggregate value based on the old

aggregate value and information from the records that are activated and invalidated. For instance, if the current maximum is USD 1900 and the value 1900 is invalidated, we need to know about the second highest active value to retrieve the new maximum. In other words, we need to keep state of historic tuples as we go along.

To that end, we use an algorithm inspired by online Skyline computation [5] and introduce a data structure which is updated incrementally: That is, at each point in time, we keep a list of Top-K values. We keep those Top-K values sorted so that we have immediate access to these values if the maximum, top two, top three, or so values are invalidated. We keep all the other activated values (i.e., the Top K+1, K+2, ... values) in a separate, unsorted vector which we call *Inserted Values*. In addition, we store all invalidated values which are not in Top-K in the *Deleted Values* vector. These unsorted values are only needed if the Top-K values are all invalidated which happens rarely if K is chosen conservatively.

Figure 6 illustrates this approach. The Top-K values are represented as an ordered multiset (in order to simplify the invalidation of identical values), backed by a red-black tree. In an experiment with real-life HANA data and queries, we set K to 0.01% of the number of distinct values in the data set and this way, almost all activations and invalidations were handled from the Top-K multiset.

Computing the `MEDIAN` requires special attention, but makes use of the same principles as `MIN` and `MAX`: Rather than keeping one Top-K list, two Top-K lists must be maintained to compute the `MEDIAN`. One list for the Top-K values below the median and another list for the Bottom-K values above the median.

Complexity. Determining the complexity for selective aggregates is more complicated since it involves an estimation of how the different parts of the Top-K data structure are used. Assuming N rows in the temporal table, each event is processed exactly once and inserted/removed into/from the Top-K data structure. For tuple activation, the new value is either added to the multiset or appended to the (unsorted) vector of values that do not make it into the Top-K. For invalidation, two possible cases may occur: 1) The value is not in the multiset. In this case, the value will be simply appended to the *Deleted Values* vector. 2) The value is in Top-K. In this case, the value is removed from the Top-K multiset. If the Top-K multiset is empty as a result of this deletion, it is rebuilt with the Top-K values from the (unsorted) vector of values that initially did not make it into the Top-K multiset. The complexity of all these operations is as follows:

1. appending to one of the vectors: constant cost, i.e. $\mathcal{O}(1)$
2. inserting or removing from the multiset: cost is $\mathcal{O}(\log K)$
3. refilling the multiset: cost is $\mathcal{O}(2 \cdot (L + H \cdot \log H) + H)$ where $L \leq N$ is the number of values in the bigger vector and H is the number of elements that are retrieved from the irrelevant vectors when the multiset gets empty. H is currently chosen empirically as $2 \cdot K$. So the cost is $\mathcal{O}(N)$ assuming the cost for a partial sort is linear for the table size.

For each tuple of the temporal table, each of the three cases described above occurs with probability $p1$, $p2$ and $p3$, respectively. Table 2 shows the values for these probabilities that we have measured for data based on a real-world scenario, generated by the TPC-H History Generator [10]. The worst-case for this algorithm in the example of a `MAX` function is a descending sequence of numbers. In this case, the cost is $\mathcal{O}(N \log H)$. It can be deduced from the resulting probabilities that the cheapest action 1) occurs in almost all the cases (99.4%). The more expensive actions 2) and 3) are rare events. For this reason, the total execution time is approximately linear with respect to the table size N in the expected case.

p1	p2	p3
99.440040 %	0.559953 %	0.000007 %

Table 2: Top-K Probabilities

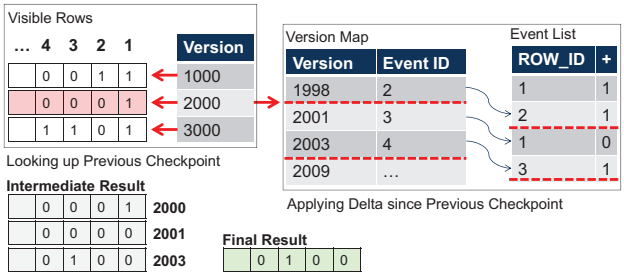


Figure 7: Time Travel to Version 2005

5.1.3 *Custom Aggregation Functions*

User-defined aggregate functions can also be supported using the *Timeline Index*. While none of the specific techniques we used for cumulative and selective aggregate functions are applicable without knowing the semantics of the aggregate function, the *Timeline Index* is nevertheless useful: In any case, it creates a window of *visible* tuples at each point in time and worst case, this window can be scanned with linear effort to construct the aggregate value. As part of future work, we intend to develop a framework that allows users to plug in efficient implementations for user-defined aggregates using a *Timeline Index*.

5.2 Time Travel

Establishing a consistent view of a previous version of the database is the most commonly used temporal operator in commercial systems. It allows the user to perform regular value queries on a single, older version of the database and corresponds closely to the *pure-timeslice* query class outlined in [20].

Example. At a given time in history, in how many cases did a product at a supplier have a stock level of less than 100 items?

```
SELECT COUNT(*)
FROM Partsupp
WHERE ps_availqty < 100
AS OF TIMESTAMP '2012-01-01'
```

For time travel, we need to establish a consistent version *VS*, i.e., provide access to exactly all those tuples that are valid for this version. As shown in Figure 7, we can achieve this by going back to the nearest previous checkpoint (if it exists) or otherwise the beginning of the *Timeline Index*. In the example of Figure 7, we use the checkpoint at Version 2000 in order to process the query that asks for Version 2005. The active set of that checkpoint is copied to an intermediate data structure. We then perform a linear traversal of the *Timeline Index* and stop when the version considered becomes greater than the version of the checkpoint, thereby covering versions 2001 and 2003 in this example. For these versions, we access the activated and invalidated *ROW_IDs* in the *Event List* and apply the changes to the intermediate data structure: 2001 invalidates *ROW_ID* 1, 2003 activates *ROW_ID* 3. Once we are finished, we can execute the query using the bit vector of the intermediate structure as a filter.

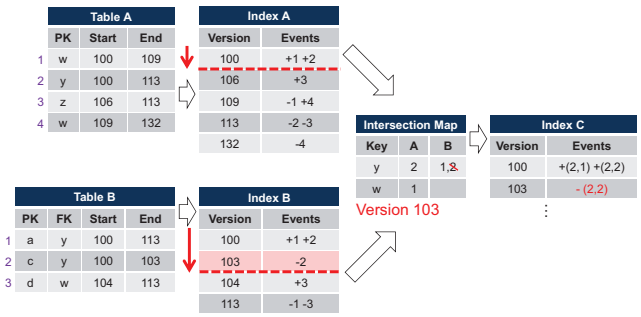


Figure 8: Timeline Join

Complexity. The cost depends on the rate at which we take checkpoints: The closer the better. Accessing a checkpoint can be done in constant or (small) logarithmic cost, as outlined in Section 4.3, whereas traversing the timeline and applying the differences is linear in the size of the *Timeline Index*. We will study the space/time tradeoffs of the checkpoint rate in Section 6.4.

5.3 Temporal Join (Timeline Join)

The third and most complex query class is the so-called *temporal join*. Just like temporal aggregation, this operator involves both the spatial dimension (i.e., the join predicate) and the temporal dimension (i.e., matching only tuples that were valid at the given point in time). In the interval-based temporal model, this means determining the interval intersection of versions.

Example. How many times did a customer with a balance smaller than 5000 have an open order with total price more than 10?

```
SELECT COUNT(*)
FROM Customer TEMPORAL JOIN Orders
WHERE o_orderstatus = 'O' AND c_acctbal < 5000
   AND o_totalprice > 10
   AND c_custkey = o_custkey
```

Our join algorithm, which we call *Timeline Join*, focuses on the temporal dimension, thereby providing most of its benefits serving temporally selective queries. It performs an equijoin on the non-temporal (spatial) attributes, making it an instance of a temporal equijoin [9]. Its output is a slightly extended *Timeline Index* for the join result, where the entries in the *Event List* are not individual *ROW_IDs* for one table, but pairs of *ROW_IDs*, one for each partner in the respective table. This design has two benefits: 1) Additional temporal operations can easily be performed on the join results, enabling temporal n-way joins (in which the *ROW_ID* pairs become n-tuples). 2) Lookup of tuples in the temporal tables (e.g., for serializing the result or applying the `WHERE` condition) can be performed in a lazy manner, i.e. late materialization. *Timeline Join* is conceptually a merge-join on the already sorted *Timeline Indexes*, augmented by a hash-join style helper structure for the value comparisons.

Figure 8 shows how the Timeline Join works. The example shows the join of two tables A and B with a composite predicate: A (spatial) value equality $A.PK = B.FK$ and time interval intersection $[A.start, A.end)$ overlap $[B.start, B.end)$. For both tables a *Timeline Index* is required. In addition, we utilize a hash-based *Intersection Map*, which relates each join key to the matching *ROW_IDs* in each table, formally IMap: $(v) \rightarrow (\{ROW\_ID_A\}, \{ROW\_ID_B\})$. To execute the join, we do a merge-join style linear scan of both *Timeline Indexes* (both ordered by *Version_ID*), using head pointers to the current row of each of the indexes. Starting from small *Version_IDs*, we advance the head pointer of the index with the lowest *Version_IDs*. When moving the head pointers we perform the following steps:

- If tuple a is activated in index A, we add its *ROW_ID* to the set for A in the intersection map, using the value of $a.PK$ as its key: IMap$(a.PK)[0] \cup (a.rowID)$.
- If tuple a is invalidated in index A, we remove its *ROW_ID* from the intersection map, using the value of $a.PK$ as key: IMap$(A.PK)[0] \setminus (A.rowID)$.

These steps are used for B in a similar fashion, using $b.FK$ as key and the second set. In our example, when we advance the head pointer for index B to version 103, we see the invalidation of the tuple with *ROW_ID* 2. Its FK value is y, so we modify the y entry in the intersection map, removing its *ROW_ID* 2 from the B set.

Changes to the Intersection Map will result in entries to the result table. Individual join partners are added or removed, yielding activation or deactivation pairs for this *ROW_ID* and its join partner. We show this case in Figure 8, where the removal of *ROW_ID* 2 for *B* at version 103 adds the invalidation pair (2, 2), since the *B* tuple *ROW_ID* 2, values (c, y, 100, 103), was joined with the *A* tuple *ROW_ID* 2, values (y, 100, 113) and now goes out of validity.

Complexity. In summary, the *Timeline Join* can be seen as a combination of a merge-join and a hash-join that are both adapted to consider temporal conditions as an extra predicate in addition to the equality of the (spatial) join predicate. The cost and complexity analysis of this join algorithm shows that it requires linear time with regard to the number of versions.

6. EXPERIMENTS AND RESULTS

This section presents the results of experiments that assess the performance of the *Timeline Index* for temporal aggregation, time travel and temporal joins. In each case, the *Timeline Index* is compared to the best-of-breed solutions from the literature. Furthermore, we compare our implementation of the *Timeline Index* with the performance of commercial database systems that support time travel queries. Additionally, this section presents measurement for index maintenance and the storage requirements.

6.1 Software and Hardware Used

All experiments were carried out on a server with 192GB of DDR3-1066MHz RAM and 2 Intel Xeon X5675 processors with 6 cores at 3.06 GHz running a Linux operating system (Kernel 3.5.0-17). Our implementation of the *Timeline Index* was integrated into an experimental database system whose design closely resembles that of the SAP HANA [7] database product: a column store that carries out query processing entirely in memory. This prototype is used inside SAP to experiment with new query processing algorithms and data structures. It is written entirely in C++.

As mentioned in Section 4.3, the only tuning knob of the *Timeline Index* is the frequency of checkpoints. This knob trades memory consumption and update performance for query speed. We studied three versions of the *Timeline Index*:

Dataset	SF_0	SF_H	|lineitem|	|partsupp|	#versions
Tiny	0.01	0.01	0.3 Mio	0.1 Mio	0.2 Mio
Small	0.1	0.1	3.4 Mio	1.3 Mio	2.2 Mio
Medium	1.0	1.0	34 Mio	13 Mio	22 Mio
Large	10.0	10.0	340 Mio	132 Mio	220 Mio

Table 3: Dataset properties

1. *No Checkpoints*
2. *Few Checkpoints:* For each table, a new checkpoint was created every 22 million versions for the *Large* dataset and every 2.2 million for *Medium*.
3. *Many Checkpoints:* For each table, a new checkpoint was created every 4.4 million versions for the *Large* dataset and every 0.44 million for *Medium*.

Unless otherwise stated, all measurements are taken with data in random physical order. For reference, we studied the performance of two commercial database systems whenever possible. The first commercial database system was the current release of HANA (without *Timeline Indexes*) and the second commercial system is referred to as *System X* because the license agreement does not allow us to reveal its true identity. Furthermore, we studied the performance of the following temporal index structures:

- *Elmasri 1990:* The Time Index as described in [6]. As no implementation was available from the authors, we implemented it ourselves. In its basic version, it only supports the time dimension. To gain better query performance, we implemented a two-level version, which uses a Time Index for every value.
- *MVBT:* The Multi-version B-tree in the Java-based XXL library[1], maintained by the authors of [3]. The MVBT provides a combined key/time index and supports a wide range of temporal queries. We tuned this implementation by using an in-memory storage container (instead of a disk-based container) and by adapting the page size for best performance. While we could measure basic index operations and time travel, no support for temporal aggregates and temporal joins is available in XXL. Unfortunately, we could not get implementations for these operations from the authors of [27] and [28]. Therefore, we used our own implementation of MVBT-based temporal joins and did not include experiments for temporal aggregation.

As additional baselines for the experiments with temporal aggregation, we used implementations of the following algorithms:

- *Snodgrass 1995:* We used our own implementation of [12], as no other implementation was available.
- *Böhlen 2006:* We used the authors' implementation of [4].

Time travel is supported natively by the following commercial database systems:

- *SAP HANA:* As a baseline, we used the release version of our database system without the implementation of *Timeline Index*.
- *System X:* We compared our results to a (traditional) general-purpose database system which is row-based.

For temporal joins, we compared the *Timeline Index* to our implementation of a traditional *hash join*.

[1]http://xxl.googlecode.com/

Table	TPC-H dbgen	Inserts	Updates	Deletes
Customer	0.2 Mio	0.2 Mio	0.6 Mio	0.0
Lineitem	6.0 Mio	1.6 Mio	1.2 Mio	0.2 Mio
Nation	25.0	0.0	0.0	0.0
Orders	1.5 Mio	0.4 Mio	0.3 Mio	0.1 Mio
Region	5.0	0.0	0.0	0.0
Partsupp	0.8 Mio	0.0	1.2 Mio	0.0
Part	0.2 Mio	0.0	0.0	0.0
Supplier	10000.0	0.0	0.0	0.0

Table 4: Operations per Table ($SF_0 = 1.0$ and $SF_H = 1.0$)

6.2 Benchmark

There is no standard benchmark for temporal databases. Only very recently, [2] proposed temporal extensions to the TPC-H benchmark. As a result, we developed our own, application-centric benchmark based on customer use cases and standard benchmarks such as TPC-H and TPC-C. The goal was to cover a broad spectrum of features of a temporal database system. A technical report [10] describes this benchmark in more detail. All our benchmark databases have a TPC-H schema (i.e., Customers, Orders, Lineitems, etc.). As a result, we can execute any TPC-H query on them. In order to create a history, our benchmark databases are generated in two steps:

- *Step 1:* Generate a Version 0 of the database. This version is generated using the TPC-H *dbgen* tool.
- *Step 2:* Generate a history by executing TPC-C transactions. Each TPC-C transaction generates a new database version. As a result, we can run time travel, temporal aggregation and temporal joins on the resulting temporal database. We adapted the TPC-C transactions to run on the TPC-H schema as described in the CH-benchmark [8].

Corresponding to these two steps, our benchmark databases are characterized by two scaling factors:

- SF_0: Scaling factor of the *dbgen* tool creating Version 0.
- SF_H: Number of update transactions applied in Step 2. An example transaction is a new customer who registers his address and places an order. As a result, SF_H determines the number of versions in the benchmark database.

Given the widely varying cost of temporal operators and specific implementations, we studied four different database scaling factors (SF_0 = SF_H): *Tiny: 0.01*, *Small: 0.1*, *Medium: 1.0* and *Large: 10.0*. These databases are characterized in Table 3. All four databases fit into the main memory of our server. This fact was exploited in the implementation of all approaches (*Timeline Index*, commercial products, etc.) so that no I/O was carried out as part of any of the experiments reported in this paper. For a better understanding of the effects of creating versions with TPC-C transactions, Table 4 shows how many inserts, updates and deletes are carried out for each table of a *Medium* TPC-H database with SF_0 = SF_H = 1.0 (i.e., 2.2 million TPC-C transactions). Note that this distribution of updates keeps the properties (such as correlations and dependencies) of dataset equal to that of a normal TPC-H dataset.

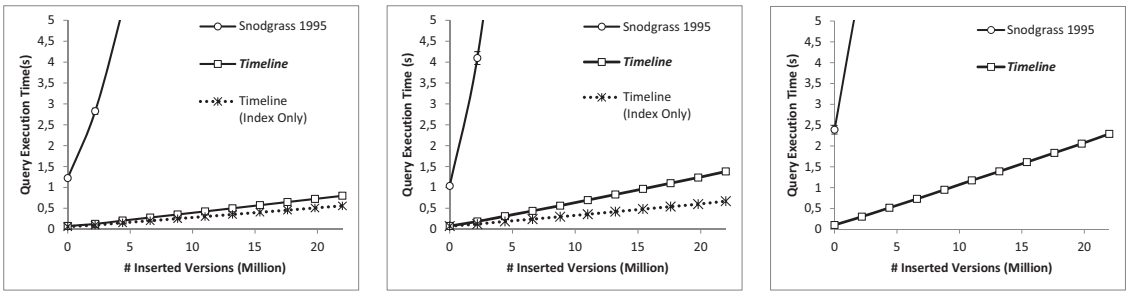
As benchmark queries, we adapted SAP customer use cases and applied them to the TPC-H schema [10]. We will describe the specific queries used in our experiments together with the experimental results in the following subsections.

6.3 Experiment 1: Temporal Aggregation

# Inserted Versions (Mio)	Base Data	0.04	0.09	0.13	0.18	0.22
Elmasri 1990	48.44	108.44	266.17	543.79	779.51	1064.76
Böhlen 2006	1.57	2.62	4.04	5.44	7.43	10.90
Snodgrass 1995	0.02	0.03	0.06	0.12	0.16	0.20
Timeline Index	*0.0006*	*0.0013*	*0.0020*	*0.0027*	*0.0039*	*0.0051*

Table 5: Temporal Aggregation: SUM [Tiny Dataset] (sec)

The first set of experiments studied the performance of the *Timeline Index* for temporal aggregation. As baselines, we used the classic algorithm Snodgrass 1995 [12] and the recently devised algorithm Böhlen 2006 [4]. Furthermore, we studied the performance of Elmasri 1990 [6] as a representative for a generic



(a) SUM, Sorted by Start Time (b) SUM, Optimal Order (c) MAX, Optimal Order

Figure 9: Temporal Aggregation [Medium Dataset]

# Inserted Versions (Mio)	Base Data	0.04	0.09	0.13	0.18	0.22
Elmasri 1990	55.39	119.05	295.26	600.42	881.98	1512.21
Böhlen 2006	8.53	19.61	54.46	102.60	149.26	198.73
Snodgrass 1995	0.01	0.03	0.06	0.11	0.16	0.21
Timeline Index	*0.0011*	*0.0025*	*0.0040*	*0.0056*	*0.0074*	*0.0095*

Table 6: Temporal Aggregation: MAX [Tiny Dataset] (sec)

Selected Version (Mio)	0	44	88	132	176	220
Elmasri 1990	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
MVBT	10.72	11.00	11.08	10.91	10.71	9.33
System X	5.32	5.61	8.21	9.86	8.86	1.14
SAP HANA	1.07	1.42	1.75	2.09	2.54	2.89
Full Scan	1.06	1.07	1.05	1.06	1.04	1.05
Timeline Index	*0.20*	*0.20*	*0.21*	*0.21*	*0.21*	*0.22*

Table 7: Time Travel for Variable Version [Large Dataset] (sec)

temporal index structure. Since the performance of the methods varied drastically, we split our results in two: 1) Figure 9 shows the results for the two most competitive methods on a *Medium* dataset; i.e., *Timeline* and *Snodgrass 1995*. 2) Tables 5 and 6 show the results of all methods on a *Tiny* dataset; all other approaches (except *Timeline* and *Snodgrass 1995*) timed out for any database bigger than *Tiny*.

Figure 9a) and 9b) shows the running time to compute a temporal aggregations according to the example of Section 5.1.1 with a `SUM`, using the *Timeline Index* and Snodgrass 1995. *Timeline Index* clearly outperforms Snodgrass 1995. The gap becomes larger when the duration of the temporal aggregation gets longer (i.e., the more tuples need to be aggregated). Comparing Figures 9a) and 9b), the difference becomes even more pronounced when the table is not ordered by the *start* field; ordering by some other criterion is important to achieve optimal compression. While the *Timeline Index* is robust and does not require temporal order, Snodgrass 1995 is particularly sensitive to the order and, thus, limits the effects of compression in a column store. In a separate experiment (not shown for brevity) we found out that ordering by *start* never achieves the best compression factor for SAP HANA.

We also include the cost of "Index Only" operations, which gives us some additional insights: 1) The (lower) cost of Index-Only operations such as `COUNT` 2) The order-independence of index operations, as the index cost is roughly the same for a) and b) 3) The moderate, but order-dependent cost of fetching from the temporal table, in contrast to the prohibitive cost of random I/O on disk.

Figure 9c) shows the results for the `MAX` temporal aggregation query in Section 5.1.2. Again, this query is, in theory, more complex to process with a *Timeline Index* whereas Snodgrass 1995 is agnostic to the aggregation function. Indeed, comparing Figures 9b) and c) the *Timeline Index* performs slightly worse for the *MAX* query, but the effects are small. Overall, the *Timeline Index* still clearly outperforms Snodgrass 1995. Even when varying the benchmark parameters and testing other queries, we could not find a single case in which Snodgrass' algorithm was better.

Since Böhlen 2006 and Elmasri 1990 did not scale well for any database bigger than *Tiny*, we present their results on the *Tiny* dataset only. Tables 5 and 6 summarize all the results. *Timeline* outperforms Böhlen 2006 by roughly two orders of magnitude and Elmasri 1990 by four.

Even though we did not experiment with this feature, *Timeline* provides an additional benefit: It is the only method that can effectively process a temporal aggregation over a limited time period; e.g., executing a temporal aggregation only for the years 2008-2010. Since the access to a specific version is fast (see next experiment), the cost for this aggregation is effectively linear to the number of versions in the query range, not in the table.

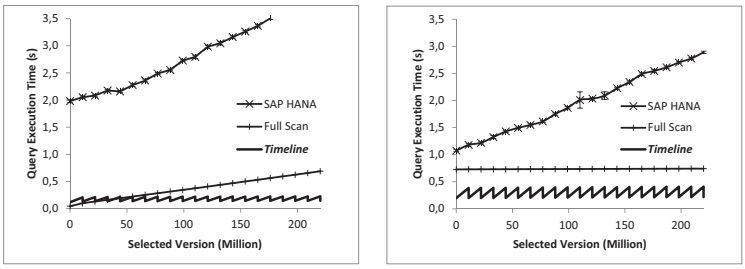
6.4 Experiment 2: Time Travel

Figure 10 and Table 7 show the performance of the *Timeline Index* for time travel queries for a *Large* dataset. Again, we split the presentation of the results for the various methods due to the huge variance in performance.

We varied the point in time that is queried: At the very left, the query is executed *AS OF* Version 0 of the database, the oldest possible version. At the very right, the query is executed against the current version of the database. We measured the *Timeline Index* with checkpoints created every 11 million versions and studied the two commercial database systems as well as the two general temporal indexes. As an additional baseline, we examined a table scan to process this query. Figure 10a) shows the results for the case that all tables are ordered by *start* time. The clear winner in this experiment is the *Timeline Index*: It performs well throughout the spectrum.

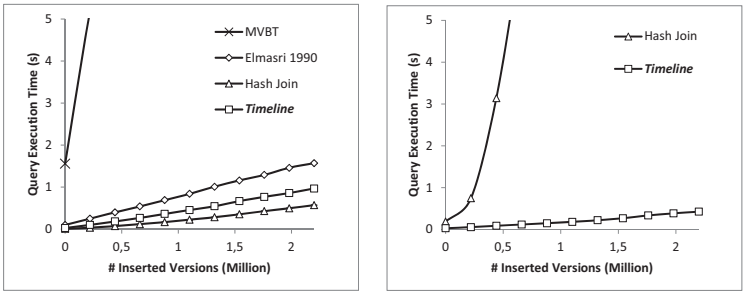
The response time of a scan-based approach grows linearly with the version number of the time travel target if the table is ordered by *start* time. With a growing version number, more and more of the table needs to be read and in an extreme case, the whole table (with all versions of all tuples) needs to be read in order to find the current version of all tuples.

The numbers for the release version of SAP HANA are significantly worse than the results that could be achieved with a scan because the cost of restoring an old transaction context exceeds the cost of a table scan. Figure 10b) shows the results for cases in which the table was not clustered by *start* time; instead, it was ordered to get the best possible compression. As can be seen, the *Timeline Index* is robust and shows (almost) the same performance,



(a) Sorted by Start Time (b) Optimal Order

Figure 10: Time Travel Query for Variable Version [Large Dataset]



(a) Space Selective Join Query (b) Time Selective Join Query

Figure 11: Join Execution Time for Increasing Table Size [Small Dataset]

independent of the ordering of the table. The additional cost due to non-linear tuple fetching is minimal. The performance of SAP HANA improves significantly because it benefits from compression, thereby reading less data from main memory into the CPU caches. The scan-based approach performs worse: Without clustering by *start* time, this approach needs to read the whole table in all cases.

Table 7 provides an overview on the remaining competitors: System X is relatively fast to access the current version, but otherwise the access time grows with the version number. It is roughly an order of magnitude slower than *Timeline*. MVBT as a temporal index gives approximately constant access time, but is also relatively slow. We omit Elmasri 1990, because its index could not be created within 24 hours for the *Large* dataset.

6.5 Experiment 3: Temporal Join

In this set of experiments, we studied the performance of using the *Timeline Index* to process temporal joins. As a baseline, we used a regular hash join. The performance of a temporal join depends on two factors: (a) *spatial selectivity*, which determines how many tuples of each relation match regardless of the temporal dimension and (b) *temporal selectivity*, which determines the relation sizes for each version. Putting it differently, a temporal join is a two-dimensional join, where selectivity in both dimensions matters.

To test temporal joins with varying selectivity, we studied two different join queries. First, we studied the temporal join query of Section 5.3. This query is highly selective in the spatial dimension. Figure 11a) shows the results for this query. Then, we studied the following query which is less selective in the spatial dimension and, thus, relatively more selective in the temporal dimension:

```
SELECT COUNT(*)
FROM Orders TEMPORAL JOIN Lineitem
WHERE l_returnflag = 'A'
  AND o_orderstatus = l_linestatus
  AND o_totalprice < 2500
```

# Inserted Versions (Mio)	Base Data	4.4	8.8	13.2	17.6	22.0
Elmasri 1990	T/O	T/O	T/O	T/O	T/O	T/O
MVBT	39.0	51.9	92.1	121.7	165.3	223.8
Böhlen 2006	13.0	25.4	42.0	64.0	87.7	114.0
Timeline - many checkpoints	0.1	1.5	3.4	5.2	7.3	9.5
Timeline - few checkpoints	0.1	1.5	3.4	5.1	7.2	9.3
Timeline - no checkpoints	0.1	1.4	3.2	4.9	6.9	9.0

Table 8: Index Construction Time (sec) [Medium Dataset]

Figure 11b) shows the results for this query. In Figure 11a), it can be seen that a traditional hash join is unbeatable if the query has a high selectivity in the spatial dimension. As Elmasri 1990 creates a tree of keys and for each key a tree of all versions, spatial selectivity can be exploited well by this data structure resulting in a performance similar to hash join. Nonetheless, *Timeline* is also very competitive, within a small constant factor of the hash join. The performance of MVBT is somewhat unsatisfactory, as we could only rely on an index-nested loop join, and not on the fully optimized joins proposed in [27].

In turn, as shown in Figure 11b), *Timeline Join* is the best choice if the selection along the temporal dimension matters. This result agrees with the outcomes of all other experiments: The *Timeline Index* is a great way to carry out any kind of selection in time. In contrast, both MVBT and Elmasri 1990 time out for this query because they rely on a space selective predicate.

6.6 Experiment 4: Index Construction and Maintenance

The time for constructing a new *Timeline Index* data structure is shown in Table 8. We measured the time for a complete index construction for the LINEITEM table with variable size and we compared the results to other index structures. The time for constructing an index for Elmasri 1990 was more than one hour already with the *Small* dataset. Also for MVBT, index construction

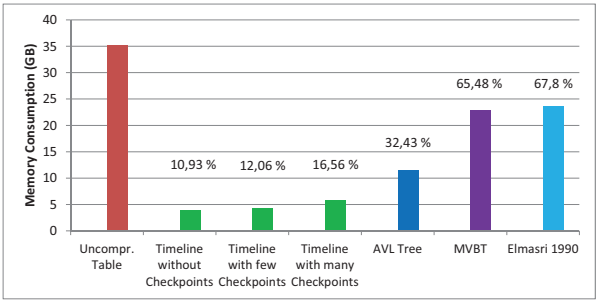


Figure 12: Memory for the LINEITEM Table [Large Dataset]

is very expensive. As shown by the measurements, the time for constructing the *Timeline Index* is linear with respect to the table size and much faster than an AVL tree used by Böhlen 2006. The construction of a *Timeline Index* is very efficient, so it is even feasible to construct the index lazily, e.g., with the first execution of a temporal operator. Checkpoints put only a minimal overhead on index construction.

Updates are supported well by *Timeline Index* by incrementally appending events to the index. Yet, checkpoints result in additional moderate costs. For space reasons, we omit the graph.

6.7 Experiment 5: Memory Consumption

The last experiment shows the memory consumption of the *Timeline Index* and its competitors. We measured the memory consumption for the LINEITEM table on the *Large* dataset. As a comparison, we show the memory consumption of the uncompressed temporal table for LINEITEM, which is 35.2 GB. For this table, the size of the *Timeline Index* is 3.8 GB, which is approximately 10% of the table memory consumption. The size of one checkpoint is 40.5 MB, which is rather small because it is a single bit vector. Therefore the memory consumption only slightly increases for few checkpoints. For many checkpoints the memory consumption of the index data structure is still only 17% of the table which is much smaller than the memory required for MVBT and Böhlen 2006. MVBT has to deal with replicated entries, if they span active and outdated pages. The memory consumption of Elmasri 1990 is very high because of the replication of data for different versions.

7. CONCLUSION

This paper presented a novel, versatile index structure for temporal tables called *Timeline Index*. The *Timeline Index* is universal, thereby supporting a large variety of temporal operators. It is space-efficient; typically the size is only a small percentage of the size of a temporal table and a single *Timeline Index* per temporal table is sufficient. It is flexible and does not limit other decisions of the physical design such as compression. Furthermore, it integrates nicely into an existing database system, thereby taking advantage of highly optimized code paths to scan data, parallelize queries, and works well on modern (NUMA) hardware. Temporal operators can be nested and an efficient result construction can be achieved by late materialization. The performance is predictable, with only a single tuning knob, the number of checkpoints. Most importantly, the *Timeline Index* is fast: It beats all best-of-breed approaches in all our performance experiments with an in-memory column store; in some cases by orders of magnitudes.

Currently, the most important line of future work is to apply the *Timeline Index* to *application time* in addition to *system time*. This way, the *Timeline Index* would become applicable to a full bi-temporal data model.

Acknowledgments

We thank our colleagues at SAP, Andreas Tonder, Ingo Müller and Jonathan Dees for their valuable feedback and comments on our work. We would also like to thank Michael Böhlen and Bernhard Seeger (as well as their respective research groups) for providing access to their index implementations.

8. REFERENCES

[1] G. Adelson-Velskii and E. M. Landis. An Algorithm for the Organization of Information. In *Proceedings of the USSR Academy of Sciences*, 1962.

[2] M. Al-Kateb et al. Adding a Temporal Dimension to the TPC-H Benchmark. In *TPCTC Workshop*, 2012.

[3] B. Becker et al. An Asymptotically Optimal Multiversion B-Tree. *VLDB J.*, 5(4), 1996.

[4] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional Aggregation for Temporal Data. In *EDBT*, 2006.

[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[6] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The Time Index: An Access Structure for Temporal Data. In *VLDB*, 1990.

[7] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[8] F. Funke et al. Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark. In *TPCTC Workshop*, 2011.

[9] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join Operations in Temporal Databases. *VLDB J.*, 14(1), 2005.

[10] M. Kaufmann, D. Kossmann, N. May, and A. Tonder. Benchmarking Databases with History Support. Technical report, SAP AG, 2013.

[11] M. Kaufmann, D. Kossmann, N. May, and A. Tonder. SQL Extension for History Tables:. Technical report, SAP AG, 2013.

[12] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *ICDE*, 1995.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[14] K. G. Kulkarni and J.-E. Michels. Temporal Features in SQL: 2011. *SIGMOD Record*, 41(3), 2012.

[15] D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *SIGMOD*, 1989.

[16] D. Lomet et al. Transaction Time Support Inside a Database Engine. In *ICDE*, 2006.

[17] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. The LHAM Log-Structured History Data Access Method. *VLDB J.*, 8(3-4), 2000.

[18] R. Rajamani. Oracle Total Recall / Flashback Data Archive. Technical report, Oracle, 2007.

[19] S. Ramaswamy. Efficient Indexing for Constraint and Temporal Databases. In *ICDT*, pages 419–431, 1997.

[20] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.*, 31(2), June 1999.

[21] C. M. Saracco et al. A Matter of Time: Temporal Data Management in DB2 10. Technical report, IBM, 2012.

[22] R. T. Snodgrass et al. TSQL2 Language Specification. *SIGMOD Record*, 23(1), 1994.

[23] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *ICDE*, 1994.

[24] I. Stoica et al. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[25] M. Stonebraker. The Design of the POSTGRES Storage System. In *VLDB*, 1987.

[26] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *VLDB*, 2003.

[27] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient Temporal Join Processing Using Indices. In *ICDE*, 2002.

[28] D. Zhang et al. On Computing Temporal Aggregates with Range Predicates. *ACM Trans. Database Syst.*, 33(2), 2008.