Storing and Processing Temporal Data in a Main Memory Column Store

Martin Kaufmann
(supervised by Prof. Dr. Donald Kossmann)
SAP AG, Walldorf, Germany and Systems Group, ETH Zürich, Switzerland
martin.kaufmann@inf.ethz.ch

ABSTRACT

Managing and accessing temporal data is of increasing importance in industry. So far, most companies model the time dimension on the application layer rather than pushing down the operators to the database, which leads to a significant performance overhead. The goal of this PhD thesis is to develop a native support of temporal features for SAP HANA, which is a commercial in-memory column store database system. We investigate different alternatives to store temporal data physically and analyze the trade-offs arising from different memory layouts which cluster the data either by time or by space dimension. Taking into account the underlying physical representation, different temporal operators such as temporal aggregation, time travel and temporal join have to be executed efficiently. We present a novel data structure called *Timeline Index* and algorithms based on this index, which have a very competitive performance for all temporal operators beating existing best-of-breed approaches by factors, sometimes even by orders of magnitude. The results of this thesis are currently being integrated into HANA, with the goal of being shipped to the customers as a productive release within the next few months.

1. INTRODUCTION

Most data sources in real-life are not static but change their information in time. This evolution of data in time can give valuable insights to business analysts. An example is the inventory of a big distributed mail order business, where bottlenecks in the supplier chain can be identified easily by computing the aggregated number of items out of stock per supplier in a time interval. This operator is referred to as *temporal aggregation*. Another example is a trader comparing the value of his investment portfolio with the state *AS OF* a year ago. Retrieving an old version of a table is called *time travel*.

Companies have a high demand to maintain and query such temporal data. Yet, only few commercial database systems provide support for the time dimension in their products. Oracle has been supporting time travel operators by means of its Flashback feature for more than 10 years. IBM DB2 [13] and Teradata have only very recently added temporal features to their products. However, the supported temporal operators in these products are limited to time travel. Moreover, all these systems store their data on a disk-based row-store.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

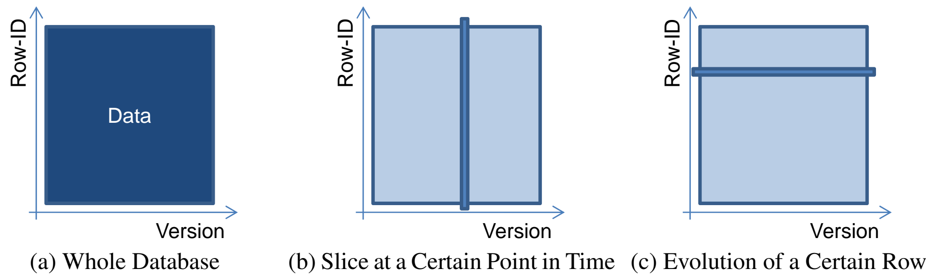
On the academic side, temporal data management has been the subject of extensive research since the 1990s. Since Snodgrass' definition of the temporal data model [14], there has been a large body of work in this area, summarized in [12, 4]. This related work covers proposals for index structures (e.g., multi-version B-trees [1]) and algorithms for certain kinds of queries (e.g., temporal aggregation [10, 2] and temporal joins [4, 15]). In most related work the focus was on disk based structures, optimizing for I/O behavior.

Driven by the superior performance of in-memory databases and the increasing amount of available memory, SAP HANA follows another approach: We keep all data (both current and temporal) in main memory. Adding distribution and compression to the picture, HANA is already able to handle temporal queries on hundreds of terabytes of data in main memory.

The topic of this PhD thesis is to investigate how support of temporal features can implemented natively in HANA. The challenge is to find a unified solution which takes advantage of modern hardware and provides optimal query execution times for the three most important temporal operators at the same time: 1) temporal aggregation 2) time travel and 3) temporal join. In addition, memory consumption should be minimized as this is an important cost factor for an in-memory database system. In this context, we address the following three questions:

(a) How to store temporal data in an in-memory column store? The first step is to study alternative approaches to represent temporal data in a main memory column store, as published in [7]. Here we focus on the physical storage of temporal data and scan-based algorithms rather than index data structures. The experiments we run on the different memory layouts give insight into the fundamental space-time tradeoffs of versioned column stores. A hybrid approach, which partially clusters the data per time and space, shows the most balanced performance for our use cases.

(b) How to index temporal data in an in-memory column store? Whereas in the previous approach we considered scan-based solutions only, we now present a novel index data structure called *Timeline Index* [9] and algorithms for processing a large variety of temporal queries based on this index. Only one instance of a *Timeline Index* is required per table, with memory consumption linear with respect to the table size. The performance results for the temporal operators are very competitive - up to several orders of magnitude faster then current results from related work in literature.



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

Figure 1: Different Dimensions of a Relation with Temporal Data

(c) How to measure performance? During our work, the performance evaluation of our operators turned out to be a problem as no standard benchmarks for temporal databases are available. We therefore propose a new *Benchmarking Service* [5], which provides an abstract and generic model of benchmarks. This service supports consolidating and unifying various benchmarks. Furthermore, it simplifies the development of new benchmarks. We use this system to micro-benchmark different implementations of temporal operators and for the development of new benchmarks such as [6].

The rest of this paper is organized as follows: We first give an overview about HANA, which is our target database system. Section 3 describes different memory layouts to store temporal data. In Section 4 the *Timeline Index* is introduced. The *Benchmarking Service* is sketched in Section 5. We give some avenues for future research in Section 6 and conclude the paper in Section 7.

2. THE SAP HANA DATABASE SYSTEM

SAP HANA [3] is a commercial database system, which consists of a combination of a main memory column store and a main memory row store. In this paper we focus on the column store engine.

2.1 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 adopted 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.

Multiple compression schemes are applied to reduce the main memory consumption and improve query execution times. In order to achieve good insert/update performance, one or multiple non-compressed delta stores receive the incoming data and later merge them to the compressed format. For consistency, all operations take all stores into account.

SAP HANA is a distributed database system which allows for the usage of multiple servers for one installation. 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. The OLTP engine is based on Snapshot Isolation for concurrency control; Snapshot Isolation is an important prerequisite to implement clear semantics for temporal data management.

2.2 Support of Temporal Data

Temporal data (transaction time) is already available natively in HANA, but only the time travel operator is supported at the moment. Transactional and analytic operations on temporal data are available in the same database system by separating current from temporal data. Current and temporal data are separated in different structures, but both are kept in main memory. 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. Both can be implemented by temporal data structures.

3. STORING THE TIME DIMENSION

In this section, different layouts for storing temporal data in an in-memory database system such as SAP HANA are compared.

3.1 Memory Layouts

As shown in Figure 1, there are two dimensions relevant to a relation that contains historical data: the time dimension (i.e., slice the relation to show the 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). Data can be clustered along no more than one of these two dimensions at the same time. Depending on this decision, different costs have to be paid for the temporal operators. In this section, we focus on temporal aggregation and time travel.

Clustering by Row. In the *clustering by row* approach shown in Figure 2, space for a fixed number of versions is reserved for each row. 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 as in the case of a traditional column store. val_{im} is the value of row i which was 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 given in Figure 2 we consider a versioned *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*.

Clustering by Version. In the *clustering by version* approach, for each version of a row four values are stored in an array: The

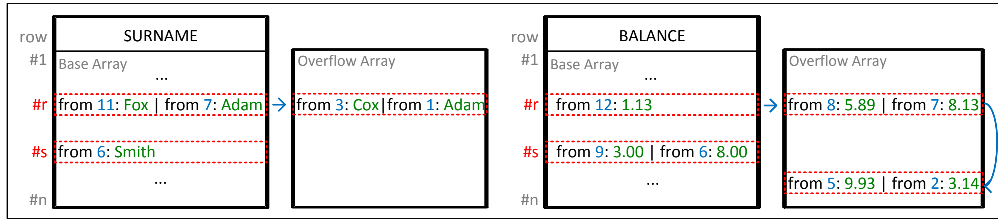
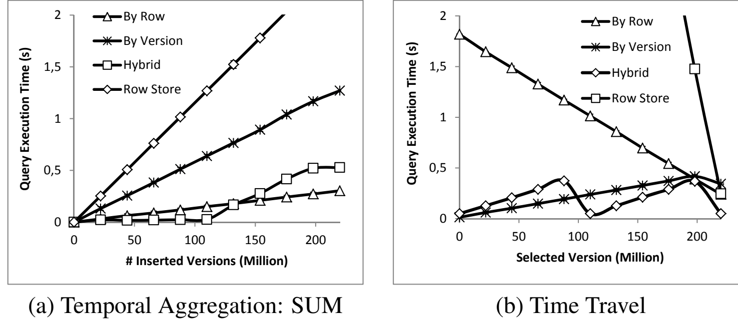


Figure 2: Physical Representation of Temporal Data Clustered by Row



(a) Temporal Aggregation: SUM (b) Time Travel

Figure 3: Temporal Operators Executed for Different Memory Layouts

Row-ID i, the value val and a half-closed 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 was 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').

Hybrid Clustering. The layout of the *hybrid* approach is similar to the *clustering by version* layout, but it includes additional *checkpoints*, each containing the latest version for all rows at the time that the checkpoint was computed.

Again, if a 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* according to the *clustering by version* approach. The tuples are therefore clustered by version. After a fixed number of updates (defined by the *checkpoint interval* parameter) 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, the data within a checkpoint is clustered by row. For keeping track of the versions for which a checkpoint is available, an index is introduced.

3.2 Measurements

For the evaluation of the memory layouts we use a data generator based on TPC-H with additional update scenarios to create a realistic workload containing a history of data as described in [6].

Temporal Aggregation. Figure 3a shows the results for computing a temporal aggregation for different memory layouts. A row store is shown as a baseline. Figure 3a shows the execution time of a query calculating the aggregated sum of *l_extendedprice* grouped by time within the version interval [60 M, 90 M].

The execution time of the *clustering by row* layout increases linearly with the number of versions because all versions of a row are clustered together and can be read sequentially. For the *clustering by version* approach, a full table scan is required to retrieve all versions of a row. The *hybrid* approach benefits from the checkpoint at version 50 M, which results in a linear scan within the interval [50 M, 90 M] only. The query execution time for the row store is the worst because of the pointer chasing for each version.

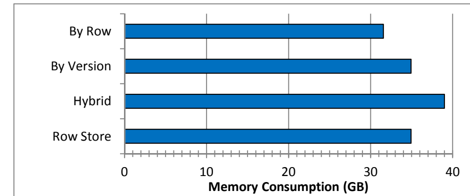


Figure 4: Memory Consumption of the Lineitem Table

Time Travel. In the next experiment shown in Figure 3b we measure the performance of a time travel operation in which the maximum value of the *l_quantity* attribute for all rows at a given version is calculated. For the *clustering by row* layout, the query execution time decreases for higher versions. This is due to the fact that the newest versions are stored in the leftmost *segment*. The execution time increases for later versions in the *clustering by version* approach because more tuples have to be scanned for a higher version. The sawtooth shape of the *hybrid* approach is caused by the execution time increasing linearly with respect to the distance to the nearest checkpoint. In addition, the latest version can always be retrieved in constant time from the *current* checkpoint. For a better visualization of the effects, only one intermediate checkpoint was created for the measurements. The performance of the *row store* decreases significantly for lower versions because a pointer has to be followed for each version.

Memory Consumption. Figure 4 shows the total memory consumption of the *lineitem* table for different layouts. The *clustering*

Name	Balance
Carl	$100
Ellen	$700

(a) Current Table

Row-ID	Name	Balance	Start	End
1	Alice	$200	101	103
2	Bob	$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 5: Example Current and Temporal Tables

Version Map

Visible Rows	Version	Event-ID
1	101	1
1, 2	102	2
2, 3, 4	103	5
2, 3, 4, 5, 6	105	7
2, 3, 5	106	9
3, 5	107	10

Event List

Row-ID	+
1	1
2	1
1	0
3	1
4	1
5	1
6	1
4	0
6	0
2	0

Figure 6: Timeline Index for the Temporal Table in Figure 5b

by row approach is more memory-efficient than *clustering by version* because the primary key has to be stored for each row only once. The memory consumption of the *hybrid* approach is higher than *clustering by version* and depends on the number of checkpoints. For our experiments, only one intermediate checkpoint was created. The memory consumption of the row store is similar to the *clustering by version* approach.

From the experiments we can derive that the *hybrid* approach has the best and most balanced execution times for different operators. Yet, the memory consumption is too high due to the full materialization of the data for each checkpoint. Therefore, in the next section we follow another approach and represent the checkpoints as a bitmap which references the temporal table. Another disadvantage is that the operators only work if the table is sorted physically by version. We therefore introduce an index data structure which 1) restores the temporal order for a temporal table with arbitrary physical order and 2) implements checkpoints more efficiently.

4. THE TIMELINE INDEX

This section describes the data structures and basic principles of the *Timeline Index* [9].

4.1 Fundamentals and Architecture

Figure 5 shows an example of how HANA manages temporal data. A similar architecture has been adopted by DB2 [13]. For every table, HANA keeps the *current* version of the table and the whole history of all versions of the table in a separate structure. For simplification we assume in this paper, that the current version is always replicated to the temporal table. The current table provides

Version	sum	
101	$200	+$200
102	$500	+$300
103	$900	-$200+$100+$500

Linear scan

Version	Events
101	+1
102	+2
103	-1 +3 +4

Timeline Index

Lookup Balance

Row-ID	Name	Balance	...
1	Alice	$200	...
2	Bob	$300	
3	Carl	$100	
4	Alice	$500	...

Temporal Table

Figure 7: Temporal Aggregation: SUM

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 that is where the *Timeline Index* takes effect: It is an index that accelerates various temporal operators carried out on different columns of a temporal table. Yet, only one *Timeline Index* is required per table. Temporal tables and *Timeline Indexes* are the focus of this work.

4.2 Index Data Structure

Figure 6 shows the *Timeline Index* for the temporal table of Figure 5b. The idea of the *Timeline Index* is keeping track of all the *visible* rows of the temporal table at every point of time. To this end, the *Timeline Index* returns all rows that are *activated* or *invalidated* at each point in time. For instance, *Row-ID* 1 of the temporal table of Figure 5b is activated at Version 101 and invalidated at Version 103 of the database. More concretely, a *Timeline Index* consists of two data structures which are scanned concurrently to implement any kind of temporal operation: *Event List* and *Version Map*. In addition, checkpoints are used to put an upper bound to the time needed to access a certain version in the table.

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.

Version Map. The second data structure of the *Timeline Index* is the *Version Map*, which keeps track of the sequence of events that are *seen* by each version of the database; i.e., by each commit of a transaction. For instance, the *Version Map* of Figure 6 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*. 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 exploit this feature. Figure 6 shows the visible rows for each version of the database in red.

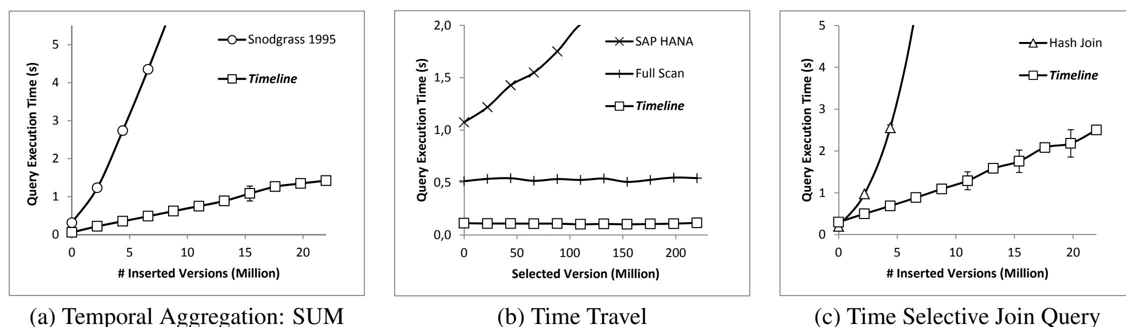
Checkpoints. Reconstructing all tuples visible at a single version requires the complete traversal of the index, leading to linearly increasing cost to access (later) versions. To overcome this problem, we augment the difference-based *Timeline Index* with a number of complete version representations at particular points in the history. We call such a full view a *checkpoint*. A checkpoint is a bit vector which represents the visible rows of the temporal table at the time the checkpoint was taken.

4.3 Index Construction

Based on the design of the *Timeline Index*, we can now describe how to efficiently create and incrementally update the data structure, even when the underlying data is not in *start* time order. The maintenance algorithms follow an approach which is based on Counting Sort. The overall cost of this algorithm is linear with respect to the size of the table since it needs to touch each tuple only twice – once for counting the number of events per version and once for writing the values to the *Event List*. Furthermore, the index can be updated incrementally by just appending the new versions and the corresponding events to the *Timeline Index*.

4.4 Temporal Operators

In this section, we describe the basic ideas of how the *Timeline Index* supports efficient processing of temporal queries for the com-



(a) Temporal Aggregation: SUM (b) Time Travel (c) Time Selective Join Query

Figure 8: Temporal Operators based on Timeline Compared to Alternative Approaches

mon temporal operator classes we found in the use cases provided by SAP customers.

Temporal Aggregation. A temporal aggregation operator (first defined in [10]) computes an aggregated value for each temporal range. For this paper, we explain temporal aggregation using a point-in-time range and a `SUM` function, which is a cumulative aggregate. For this combination, a new aggregated value can be computed directly by knowing the previous aggregated value and the change. As shown in Figure 7, we perform a linear scan of the *Timeline Index* using a single variable, *sum*, that keeps track of the aggregated value. By scanning the index, we determine the new tuples that were activated and invalidated for each version. We look up the *balance* values for all of these tuples and adjust the value of the *sum* variable accordingly for each version. Thus, the aggregated value is computed incrementally. As a result, the execution of this operator is very efficient.

The cost for temporal aggregation based on the *Timeline Index* is linear with respect to the size of the respective temporal range.

Time Travel. The time travel operator retrieves the tuples that were visible for a given version *VS*. This set of visible tuples can be computed by going back to the nearest previous checkpoint (if it exists) or otherwise the beginning of the index. Next, the active set of this 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 *VS*.

The cost for time travel is linear with respect to the distance to the nearest previous checkpoint or the size of the temporal table.

Temporal Join. The temporal join on two tables returns all tuples which satisfy a spatial predicate and whose time intervals overlap (i.e., which are valid at the same points in time). For each input table a *Timeline Index* has to be available. The output of the join operator again is a slightly extended *Timeline Index* 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. To execute the join of two tables, we do a merge-join style linear scan of both *Timeline Indexes* (both ordered by version). The result can either be materialized or taken as an input for other temporal operators (late materialization).

As the temporal join can be computed by two parallel index scans, the execution time is linear with the sizes of the input tables.

4.5 Measurements

Temporal Aggregation. Figure 8a shows the computation of temporal aggregation with a `SUM`, using the *Timeline Index* and an implementation of an algorithm published by Snodgrass in 1995 [10]. *Timeline Index* clearly outperforms Snodgrass 1995. The gap becomes larger when the duration of the temporal aggregation gets longer (i.e., as more tuples need to be aggregated).

Time Travel. Figure 8b shows the performance of the *Timeline Index* for time travel queries for varying points in time at which the query is supposed to be executed: At the very left, the query is executed *AS OF* Version 1 of the database, the oldest version. At the very right, the query is executed against the current version of the database. We measured the *Timeline Index* using 100 checkpoints compared to the current release version of SAP HANA. As an additional baseline, we studied a table scan to process this query. The clear winner in this experiment is the *Timeline Index*: It performs well throughout the spectrum.

Temporal Join. In this experiment, we studied the performance of the *Timeline Index* to process a temporal join of the *orders* and *lineitem* tables from the TPC-H schema. As a baseline, we use a regular (and highly-tuned) hash join. As shown by the measurements in Figure 8c, the query execution time for the *Timeline Index* scales linearly with the size of the temporal table. Again, the *Timeline Index* is a great way to carry out any kind of selection in the temporal dimension.

4.6 Integrating Timeline Index into HANA

The *Timeline Index* has been implemented as a prototype based on the architecture of HANA. Data structures and algorithms are designed to fit the properties of modern hardware with the goal to be integrated into the SAP HANA product. Given this perspective of a deployment in a real system, all aspects of productive software like performance, memory consumption, parallelism, and complexity of algorithms had to be taken into account. Therefore, the data structures have to be kept simple, 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 HANA column and row stores. As temporal queries are often part of OLAP workloads, however, we envision that *Timeline Indexes* are mostly used with a columnar table layout.

5. GENERIC BENCHMARKING

In this section we introduce the *Benchmarking Service* [5] motivated by our experiments in the context of temporal databases.

5.1 Modeling a Benchmark

Since the *Benchmarking Service* aims to combine flexibility with rich data operations and user guidance, a comprehensive and expressive model is required as shown in Figure 9. The key benefit of

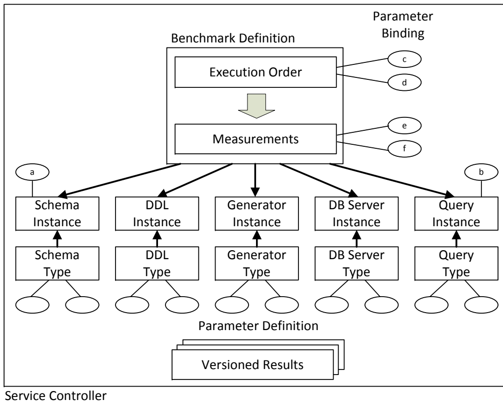


Figure 9: Data Model of the Benchmarking Service

this meta model is that artifacts (i.e., components of a benchmark) can be parameterized, stored and reused. The intuitive definition of these artifacts is achieved by a web-based UI, which also supports archiving and comparing results. Generally speaking, a *benchmark* can be seen as a subset of the cross-product of all the artifact types and parameters.

5.2 System Architecture

A distributed architecture was chosen for the service: A central *Service Controller* keeps track of the meta model instances, which includes both the actual artifacts and the results. The process of running an experiment is controlled by a *Coordinator Node*, which contains a queue of benchmarks that are about to be executed, distributes jobs and detects node failures. The benchmarks are run on several *Execution Nodes* in parallel to simulate a multi-user workload or speed up measurements. Each *Execution Node* in turn may distribute the measurements over several database servers. Database servers can be accessed at different levels, mainly using call-level interfaces such as JDBC with queries and statements derived from workloads and DBMS metadata. Besides, when necessary, scripts can be used at the OS level to start/stop databases and perform external tuning.

6. RESEARCH DIRECTIONS

There are many interesting avenues for future work in the context of temporal data in main memory column-stores. In this section we give two topics of additional projects.

6.1 Bitemporal Data

We plan to apply the *Timeline Index* also to *application time* in addition to *system time*. Supporting both time dimensions would make the *Timeline Index* suitable for the bitemporal data model, which is now part of the SQL:2011 standard [11]. The challenge is that the assumption of static data for previous versions does not hold any more for *application time*. For this reason, the append-only approach for index maintenance is not feasible in this case. An idea to tackle this problem is to employ the delta-main approach chosen for HANA also for index maintenance.

6.2 Compound Temporal Queries

Up to now we have investigated various temporal operators independently of each other. A further research direction is to consider more complex query plans including several different temporal operators per query using a *Timeline Index* also as a difference-based representation of an intermediate result rather than fully materializing tables, e.g., in the case of a temporal join.

7. CONCLUSION

This paper presented our current and on-going research in the context of providing native support for various temporal operators in a commercial in-memory column store database system.

We compared three alternative memory layouts to store temporal data physically in a column store and evaluated the tradeoffs for each temporal operator. We achieved the most balanced results for a hybrid approach, which combines both segments of data clustered by time and by space.

Yet, the requirement to physically reorganize data in favor of compression and to further improve performance was the motivation for developing a novel, universal index structure which supports a large variety of temporal operators. This *Timeline Index* is space-efficient, typically only a small percentage of the size of a temporal table and only a single instance of this index per temporal table is sufficient. Furthermore, it integrates naturally into an existing database system such as SAP HANA, thereby taking advantage of highly optimized code paths to scan data, parallelize queries, and utilize modern (NUMA) hardware. Query execution time is predictable and very 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.

A prototype with an implementation of various temporal operators based on the *Timeline Index* has already been implemented and a demo [8] has been published. We are currently working on a productive release of SAP HANA having all temporal operators (i.e., time travel, temporal aggregation and temporal join) based on the *Timeline Index*.

8. REFERENCES

[1] B. Becker et al. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4), 1996.

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

[3] F. Färber et al. The SAP HANA Database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

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

[5] M. Kaufmann, P. M. Fischer, D. Kossmann, and N. May. A Generic Database Benchmarking Service. In *ICDE*, 2013.

[6] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. TPC-BiH: A Benchmark for Bi-Temporal Databases. In *TPCTC*, 2013.

[7] M. Kaufmann, A. Manjili, S. Hildenbrand, D. Kossmann, and A. Tonder. Time-Travel in Column-Stores. In *ICDE*, 2013.

[8] M. Kaufmann et al. Comprehensive and Interactive Temporal Query Processing with SAP HANA. In *VLDB*, 2013.

[9] M. Kaufmann et al. The Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 2013.

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

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

[12] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2), June 1999.

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

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

[15] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *ICDE*, 2002.