Comprehensive and interactive temporal query processing with SAP HANA

Martin Kaufmann, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber

Angaben zur Veröffentlichung / Publication details:

Kaufmann, Martin, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, and Franz Färber. 2013. "Comprehensive and interactive temporal query processing with SAP HANA." *Proceedings of the VLDB Endowment* 6 (12): 1210–13. https://doi.org/10.14778/2536274.2536278.

Comprehensive and Interactive Temporal Query Processing with SAP HANA

Martin Kaufmann[†§], Panagiotis Vagenas[†§], Peter M. Fischer[#], Donald Kossmann[†], Franz Färber[§]

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

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

[§]SAP AG
Walldorf, Germany
franz.faerber@sap.com

ABSTRACT

In this demo, we present a prototype of a main memory database system which provides a wide range of temporal operators featuring predictable and interactive response times. Much of real-life data is temporal in nature, and there is an increasing application demand for temporal models and operations in databases. Nevertheless, SQL:2011 has only recently overcome a decade-long standstill on standardizing temporal features. As a result, few database systems provide any temporal support, and even those only have limited expressiveness and poor performance. Our prototype combines an in-memory column store and a novel, generic temporal index structure named *Timeline Index*. As we will show on a workload based on real customer use cases, it achieves predictable and interactive query performance for a wide range of temporal query types and data sizes.

1. INTRODUCTION

Managing temporal data is of ever-increasing relevance: most of the data we observe has temporal aspects (e.g., a history of updates over time). For many of SAP's customers, working in areas like business analytics (e.g., portfolio analysis, risk evaluation in banking scenarios) or compliance monitoring, there is high need to maintain and query historical data. Implementing these temporal features at the application level is cumbersome and inefficient, as our own and also IBM's [7] experience shows. SQL:2011 has finally established support for some bitemporal data management features [5] in the standard, and database vendors are slowly showing interest. For instance, IBM [7] and Teradata have recently begun adding temporal features to their products. On the academic side, a large number of data structures and operators have been designed, but only a small amount of that work has found its way into the products. Furthermore, given the time frame in which that research was performed (mostly 1990s), the focus was on disk-based structures, optimizing for I/O behavior. Given the obvious performance benefits of main memory databases [6] and the constantly increasing amount of RAM affordable to customers (cluster installations of SAP HANA already work in setups with 100s of TBs of RAM), we provide a different solution: All historical data is kept in a main memory column store, while temporal operations are supported by a novel index data structure, called *Timeline*, which can be used to process a large variety of temporal queries. In [4], we describe this index structure in detail, whereas in this work we demonstrate how it can be applied to a variety of workloads and use cases; even to some which it was not explicitly designed for. The performance is very competitive – up to several orders of magnitude faster than related work. For the scope of this demo, we show that our temporal database prototype provides query results interactively for data sets fitting into the RAM of a single machine.

2. BACKGROUND AND USE CASES

This section describes how temporal data is expressed in our system. We describe three important query classes which have significant relevance for real-life temporal query workloads, but only limited support – even in SQL:2011 and state-of-the-art DBMS.

2.1 Temporal Model

Whereas traditional databases store data in its current version only, temporal databases also keep track of the information which was visible in the system at previous points in time. In such a temporal database system, an update of an existing tuple is implemented as an invalidation of the current value and an insertion of a new version. In our work, we use the implicitly maintained System Time, as introduced by Snodgrass in [8]. An extension towards Application Time and the full bi-temporal model is an active research direction. The temporal dimension is expressed in SQL:2011 by defining two attributes which represent the *start* and *end* time for each version of a tuple, as shown in Figure 1.

2.2 Time Travel

Time Travel establishes a consistent view of a (past) state of a table. It allows the user to perform regular value queries on a single, usually older version of the data returning the tuples which were visible in the system at a given point in time. As an example, an analyst might be interested in the value of his stock portfolio as of August 1st, 2012. Time Travel is currently the most widespread use case for temporal queries and is supported by several commercial database management systems such as Oracle and DB2. Within SQL:2011, the Time Travel operator has been standardized by means of the *AS OF* clause.

2.3 Temporal Join

A Temporal Join of two tables returns the tuples for which predicates in both value and time domain are satisfied. In addition to the value condition of a traditional join, the time dimension is added to a Temporal Join: Tuples match if a) their value predicate is fulfilled and b) their time intervals overlap. The semantics of the temporal condition is that the tuples were valid at the same time. For example, we retrieve all orders that customers placed last year while living in New York (customers may have lived in another place earlier). SQL:2011 provides rudimentary support for such joins by placing an explicit join condition on the temporal columns, but does not offer any advanced expressions for temporal joins.

2.4 Temporal Aggregation

In contrast to Time Travel, which goes back to a single point in time, Temporal Aggregation considers a sequence of versions to compute an aggregated value for a temporal range. This aggregated value can be evaluated using different aggregation functions such as `SUM` and `MAX`. Some aspects of Temporal Aggregation are well-explored in terms of dedicated data structures and algorithms [1, 9]. The support in SQL:2011 is limited to simple cases in which a single aggregation period is explicitly specified by selection predicates on the *start* and *end* columns. Given this restricted support and the large number of use cases requiring more expressive semantics, we have opted for an extension of the SQL syntax and semantics, providing explicit, higher-level control over the aggregation parameters. For this extension, we need to consider two orthogonal aspects: defining the *temporal range* and the *aggregation locality*.

2.4.1 Temporal Range Selection

Many queries such as the value of an inventory for each version or the average number of pending orders for each week require repeated aggregations over time. Currently, no formal classification of such temporal ranges exists, and only the simplest case (temporal grouping [2]) has been investigated in detail. Given the task of defining ranges over data ordered by time, we utilize the window concepts from data stream systems [3] and define four types of temporal ranges: (1) **Point in Time.** (i.e., temporal grouping [2]). The aggregate is computed for each version or point in time. (2) **Tumbling Window.** The time-intervals are non-overlapping (e.g., the number of all orders shipped per calendar week). (3) **Sliding Window.** The intervals are overlapping (e.g., the value of all orders shipped within the previous 7 days, computed each day). (4) **Landmark Window.** The windows are overlapping, but have the same start point (e.g., the number of orders shipped up to each day for this year).

2.4.2 Aggregation Locality

On top of these temporal ranges we can apply a number of aggregation functions. The first, most common approach of aggregate computation only depends on versions within a specified range. In this type, which we call *range-local*, we compute aspects such as the maximum inventory of a specific item during a period of time.

The second approach of aggregate computation is specific to temporal workloads, but has so far not been investigated much. Instead of considering only the version within the target range, the "history" of a row up to the end of the target temporal range is required. We refer to this use-case as a *Non-local Aggregate*. This concept is best explained with examples: a) We are interested in the point in time at which a state change occurs (i.e., an order is marked as shipped) and b) we want to measure the state duration (i.e., how long the order was in the state "unshipped"). Since for many typical applications, states of data items are stored (like "unshipped" or "shipped"), but not state changes, we need to inspect previous versions for both cases.

The concrete syntax proposals for Temporal Aggregations and Temporal Joins are currently undergoing evaluations at SAP.

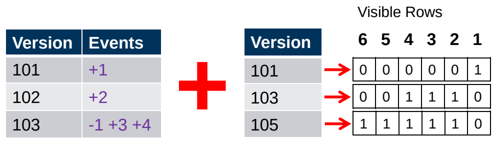
Name	Balance
Carl	$100
Ellen	$700

(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 1: Example Current and Temporal Tables



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

Version	6	5	4	3	2	1
101	0	0	0	0	0	1
103	0	0	1	1	1	0
105	1	1	1	1	1	0

Figure 2: Timeline Index

3. SYSTEM DESCRIPTION

In this section, we outline the prototype's architecture and give insights into data structures and algorithms of the *Timeline Index*.

3.1 System Architecture

Our prototype is modeled after SAP HANA, a main memory column store database which exploits large cluster setups and compression to handle large-scale data. HANA not only supports analytical workloads, but also works well for update-heavy, transactional scenarios, utilizing a delta-main merge approach.

For the management of historical data, current and temporal data are separated in different structures; temporal tables contain additional columns for the validity intervals as outlined in Figure 1. The prototype extends the main memory column store with a special index data structure (a *Timeline Index* [4]) to improve the efficiency of temporal operations. It contains different implementations of temporal operators (with and without making use of the *Timeline Index*). Our prototype is currently working on a single node until we have completed the scale-out for the temporal data structures.

3.2 Timeline Index

The *Timeline Index* is a unified index data structure which supports various temporal operators and different attributes. Only one index is required per table (see Figure 2). The idea of the *Timeline Index* is to keep track of all the visible rows of the *Temporal Table* at every point of time. This is achieved by recording the *ROW_IDs* of invalidations (i.e., deletions) and activations (i.e., insertions) for each version, in version order. By scanning this information, operators can determine the changes between versions and/or establish the set of active tuples for a specific version. Using *Timeline* as a secondary index (by storing the *ROW_IDs* of the tuples) decouples the value storage from temporal aspects. As a result, the values in the temporal tables can be stored in any order, allowing for better compression and partitioning.

The reconstruction of all tuples of a single version still requires the complete traversal of the index. 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, called *checkpoints*. By controlling the number of checkpoints, an administrator can perform a trade-off between query cost and storage overhead. As we show in [4], the space overhead of this index in minimal. Index creation and index maintenance are very fast, in the order of (a few) seconds for tables with several hundred million rows.

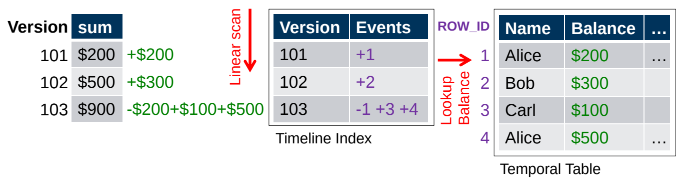


Figure 3: Temporal Aggregation: SUM

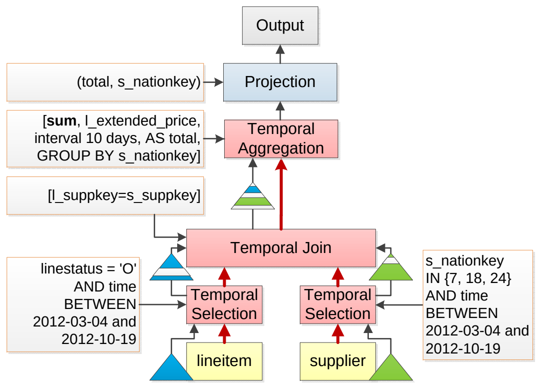

Output
Projection
(total, s_nationkey)
[sum, l_extended_price, interval 10 days, AS total, GROUP BY s_nationkey]
Temporal Aggregation
[l_suppkey=s_suppkey]
Temporal Join
linestatus = 'O' AND time BETWEEN 2012-03-04 and 2012-10-19
Temporal Selection
Temporal Selection
s_nationkey IN {7, 18, 24} AND time BETWEEN 2012-03-04 and 2012-10-19
lineitem
supplier

Figure 4: Plan of a Temporal Join and Aggregation Query

3.3 Temporal Operators

In this section, we outline how the *Timeline Index* supports efficient temporal query processing for the common temporal operator classes we found in our use cases, sketching the general ideas of [4] and describing the extensions developed since then.

Time Travel. The Time Travel operator retrieves those tuples that are visible for a given version *VS*. This set of visible tuples can be computed by going to 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 if no checkpoint exists.

Temporal Join. A Temporal Join returns all tuples from two tables which satisfy a spatial predicate and whose time intervals overlap (i.e., 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 is itself a *Timeline Index*, slightly extended, in the sense that its entries refer to 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 scan of both *Timeline Indexes*, yielding cost linear with respect to the temporal range of the inputs. The result can either be materialized or taken as an input for other temporal operators.

Temporal Aggregation. A Temporal Aggregation operator selects the temporal ranges and applies an aggregate function on them, exploiting specific properties of the aggregation to achieve optimal performance. To explain how *Timeline* supports Temporal Aggregation, we use a point-in-time range and a `SUM` function, which is a cumulative aggregate. For this combination, a new aggregate value can be computed directly by knowing the previous aggregate and the change. Using a single variable, *sum*, that keeps track of the aggregate value, we perform a linear scan of the *Timeline Index* determining the new tuples that were activated for each version and the tuples that were invalidated. In the example shown in Figure 3, we look up the *balance* values for all of these tuples and adjust the value of the *sum* variable accordingly for each version (subtract the *balance* for the invalidated tuples and add the value for the activated tuples). As a result, the execution of this operator is very efficient, especially if the overall aggregation range is bounded by a temporal ranges. In this case, we can exploit checkpoints. More complex aggregate functions introduce a modest overhead, since more state needs to be kept.

A first important extension beyond [4] is using flexible and overlapping temporal ranges, as described in Section 2.4.1. *Timeline* can still be exploited in the same manner as before since these overlapping ranges can be derived from the version-ordered activation/invalidation data in *Timeline*, utilizing the same efficient access patterns. Keeping track of the individual aggregation state and sharing some of the computation on overlapping ranges incurs additional complexity. A second extension is the support of non-local aggregates. In these cases, a full scan of the index is always needed, since both state change and state duration require determining the preceding state change. This information is not found necessarily in the (target) temporal range, but in possibly any previous area. Checkpoints are not beneficial in this case. However, the index scan is still fast and predictable.

Temporal Selection. In order to support generalized temporal query processing (see next section), we introduce a temporal selection operator in addition to the operators defined in [4]. Like a Temporal Join, it works on top of *Timeline*-indexed temporal tables and applies temporal as well as value-based predicates. The result is also a *Timeline Index* (containing the *ROW_IDs* of the tuples fulfilling the selection criteria), providing composability and the means for late materialization.

3.4 Generalizing Temporal Query Processing

Given the range of operators described in the previous section, we have all the means not only to support queries with a one-to-one correspondence to the operators, but to freely combine the operators towards generic, yet effective temporal queries. We study several classes of temporal operators with different properties: Temporal Selections and Temporal Joins maintain the temporal nature (including a *Timeline Index*) of the data, allowing them to serve as an input for any temporal operator. Temporal Aggregations and Time Travel translate temporal data into regular, non-versioned data, which can be processed by regular operators of the relational algebra. Figure 4 shows an example how these operators can been arranged in a complex temporal query plan (based on a temporal query discussed in Section 4), combining Temporal Selections, Join and Aggregation followed by non-temporal operators. As noted, all temporal operations are supported by *Timeline Indexes*, ensuring efficient execution. Temporal operations are typically placed towards the leaves of the query plan, for accessing and processing *Timeline*-indexed temporal tables. Given their generic nature, these operators can participate in query optimization.

4. DEMO

4.1 Overview and Setup

We plan to demonstrate our prototype in two ways: 1) Interactive, graphical temporal data exploration on a range of predefined, real-life workloads and 2) In-depth investigation and customization of generic, comprehensive temporal queries.

Figure 5 shows the demo setup: We run a single-system installation of the prototype on a standard laptop with 32 GB RAM and an Intel Core i7 processor. The prototype is connected to a Servlet container hosting the web GUI. We use an Ajax-based web-interface to guide the data exploration, visualize the query results (using Highcharts[1]) and display the history of response times, along with a table mapping the measurement number to the respective query parameter settings.

4.2 Interactive Temporal Data Exploration

The first part of the demo focuses on interactive data exploration using real-life queries on a temporal data set. This data set corresponds to a real customer application for inventory management. Including all relevant historical data, it consists of 25 GBs of data, fitting well into the memory of a slightly above-standard desktop or high-end laptop. We also keep scaled-down versions (100 MB, 1 GB, 10 GB) to demonstrate how the data size affects the response time. This kind of application targets individual analysts (or small teams) who wish to explore temporal properties of the data. Fast execution is necessary for typical exploration patterns like (temporal) drill down or temporal shifts to compare time periods or discover trends. The predefined queries cover a wide range of temporal analysis aspects based on real-life workloads, out of which we list some: 1) Time Travel with conventional selections, joins and aggregations to identify the low-on-stock items for certain suppliers at a given point in time, 2) Range-local Temporal Aggregations with Temporal Joins to trace the evolution of open orders per country over time (shown in Figure 6, corresponding to the query plan in Figure 4), and 3) Non-local Temporal Aggregations to examine the order processing time over time for certain regions.

For all those queries, we achieve interactive, sub-second response times for a full 25 GB data set, with graph rendering times often exceeding query processing times. This expressive and efficient temporal query support within the database is important due to several reasons: The data volumes are too big to be handled on the client and most of the explorations, while simple from a visualizations side, require complex temporal operations on a large state, which is only possible in a DB server, but not in the GUI.

To provide more freedom for exploration, users may not only choose which predefined queries to run, but also interactively adapt aspects like data size, version range selection, temporal ranges, etc. These changes are immediately reflected by the execution and visualization. We provide the history of query response times along with the parameters to show how modification of on them (e.g., a larger data set or different selection parameters) affects the execution speed.

4.3 Designing Generic Temporal Queries

In the second part, we give the user insights into generic temporal query plans and their customization. Starting from a template query plan, together with the audience, we customize the plan in different ways: Source tables for temporal operators can either be chosen from a temporal table in the application domain (e.g., *orders* table) or the result of a Temporal Join or Temporal Selection. For each of these sources, we can assign a Temporal Selection, which applies

[1]http://www.highcharts.com/

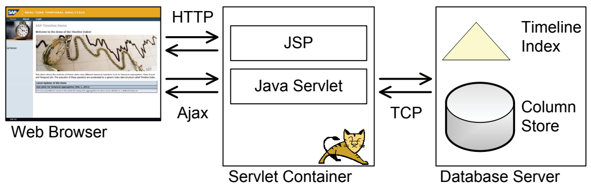
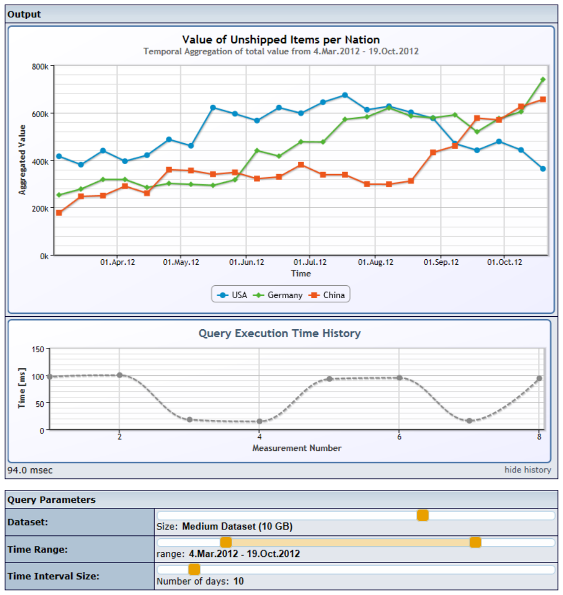


Figure 5: System Architecture of the Timeline Demo



Output

94.0 msec

Query Parameters	
Dataset:	Size: **Medium Dataset (10 GB)**
Time Range:	range: **4.Mar.2012 - 19.Oct.2012**
Time Interval Size:	Number of days: **10**

Figure 6: Screenshot of the Web Front-End

a filter with configurable temporal and non-temporal predicates. For the Temporal Join both the space and time correlation criteria can be altered, affecting the respective selectivity. For Temporal Aggregation, the aggregation function and attribute, as well as the type and size of the temporal range are possible parameters. To complete these customizations, standard relational operators can be applied to the result of the temporal operators.

5. CONCLUSION

In this demo we present a highly efficient implementation of several temporal operators in a prototype main memory database derived from SAP HANA. This implementation utilizes a column store in combination with a novel temporal index data structure called *Timeline Index*. Building on this foundation, the temporal operators can be used to answer complex, real-life analytical queries, covering operations on both the temporal and value domain. Performance is very competitive, providing interactive response times for all queries on data sizes up to 25 GB.

6. REFERENCES

[1] M. H. Böhlen et al. Multi-dimensional Aggregation for Temporal Data. In *EDBT*, 2006.

[2] J. Clifford et al. On temporal grouping. In *Temporal Databases*, 1995.

[3] L. Golab et al. On Indexing Sliding Windows over Online Data Streams. In *EDBT*, 2004.

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

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

[6] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. IT Pro. 2011.

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

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

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