Bi-temporal Timeline Index: A Data Structure for Processing Queries on Bi-temporal Data

Martin Kaufmann [#§1], Peter M. Fischer [*2], Norman May [§3], Chang Ge [†§4], Anil K. Goel [§5], Donald Kossmann [#6]

[#]*Systems Group*
ETH Zurich, Switzerland
[1]martinka@inf.ethz.ch, [6]donaldk@inf.ethz.ch

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

[§]*SAP AG*
Walldorf, Germany and Waterloo, Canada
[3]norman.may@sap.com, [5]anil.goel@sap.com

[†]*University of Waterloo*
Waterloo, Ontario, Canada
[4]c4ge@uwaterloo.ca

***Abstract*—Following the adoption of basic temporal features in the SQL:2011 standard, there has been a tremendous interest within the database industry in supporting bi-temporal features, as a significant number of real-life workloads would greatly benefit from efficient temporal operations. However, current implementations of bi-temporal storage systems and operators are far from optimal. In this paper, we present the Bi-temporal Timeline Index, which supports a broad range of temporal operators and exploits the special properties of an in-memory column store database system. Comprehensive performance experiments with the TPC-BiH benchmark show that algorithms based on the Bi-temporal Timeline Index outperform significantly both existing commercial database systems and state-of-the-art data structures from research.**

I. Introduction

Many database applications include two distinct time dimensions: The *application time* refers to the time when a fact was valid in the real world, such as duration of a contract. In addition, the *system time* represents the time when a fact was stored in the database system in order to preserve this information for audits and legal aspects. These dimensions can also evaluated in combination, such as computing the estimated total value of all contracts valid for December 2014 based on the knowledge stored in the database in June 2014.

Applications utilizing these time dimensions (bi-temporal) are not a fringe area, but there is actually much demand for effective and efficient support, as shown in a survey that we performed on SAP's application stack [1]. Given that this software covers a broad range of business tools (ERP, SCM, Data Warehousing) in many different domains (Banking, Logistics, Manufacturing), this survey covers a representative space of applications, not just SAP-specific cases. The majority of use cases require at least one application time dimension in addition to system time; some of them need up to five different application time dimensions. Access to this data ranges from conceptually simple timeslice operations to complex analytical queries, tracing the evolution of data and supporting domain specific tasks such as liquidity risk management.

However, many temporal operations are currently implemented within the application code (which is both inefficient and error-prone [2]), since there is no comprehensive and efficient support of bi-temporal data in today's DBMSs. Several big database vendors (Oracle, IBM [2], Teradata [3] and also SAP) have recently begun adding bi-temporal features to their products, following the inclusion of basic bi-temporal operations in SQL:2011 [4]. Yet, as our recent in-depth study of these systems shows [5], [6], the implementations are mostly concerned with the syntax and overall temporal semantics and fall short for efficiency by relying on non-temporal storage, indexing and query execution methods.

A likely reason for this problem is that managing bi-temporal data is a difficult task, for which the research community has not yet provided a convincing answer. Covering the various dimensions of time and possibly (key) values shares some common traits with multidimensional indexing, but many temporal operations require efficient support on a temporal order of the data [7]. The time dimensions are not symmetric, since application times may see updates to "past" data items, while system time has an append-only update behavior. Different degrees of active and outdated data items, as well as varying validity intervals, render typical space-minimizing partitioning strategies ineffective and require complex data layouts and algorithms [8], [9], [10]. Several data structures for one-dimensional temporal workloads provide strong theoretical guarantees [11], but applying them to the more general bi-temporal case has not yielded strong results.

Furthermore, given the time frame in which that research was performed (mostly 1990s), the focus was on disk-based structures optimization for I/O behavior. The system landscape is currently changing dramatically towards main memory databases due to their significant performance benefits [12] and the affordability of large amounts of RAM. In a recent work [13], we showed that even for the fairly well-studied system time dimension, a temporal index structure (called Timeline Index) specifically designed for modern hardware

with copious amount of main memory beats the existing work by a significant margin. Not being bound by I/O optimizations permits a drastic simplification in the index design as well as better support for modern hardware. In this paper we focus on in-memory column stores, even though the concepts are also applicable for disk based row stores with different trade-offs.

We apply these lessons to the more general and challenging case of bi-temporal data and tackle the set of problems stemming from the additional dimensions. The design is driven by a number of key insights: (1) Effective access to the temporal order(s) is crucial for many analytical applications (e.g., temporal aggregation), while selectivity over many dimensions is a much less relevant concern. Often, even expressions that are fairly selective in the temporal domain still yield a large number of results, limiting the benefit of indexes. Likewise, access to the history of individual tuples is rather rare. (2) Most temporal operations have a dominant dimension, while correlations over all temporal dimensions are fairly rare. We therefore rely on an index design that uses one-dimensional temporal indexes for each dimension instead of a multi-dimensional index. More specifically, we use Timeline Indexes for each dimension, in which a single system time index is maintained, while complete application time indexes are only kept at selected snapshots. Queries requesting values between snapshots use the most recent snapshot and the delta between those snapshots to reconstruct the state for all relevant time dimensions. As our results show, computing these deltas is possible at moderate overhead, but additionally the index is much more compact than any other competing approach.

In summary, this work makes the following contributions:

- a novel main memory index capable of supporting a wide range of temporal operations on bi-temporal data,
- index maintenance algorithms that can trade off space consumption, update cost and query performance,
- uniform implementations for temporal operations, regardless of the time dimension, and
- a performance analysis of the index and operators, showing their performance and comparing them to existing methods and systems for bi-temporal and spatial data.

This paper is structured as follows: Section II provides a general overview of the state-of-the-art of (bi-)temporal data management with a special focus on indexing approaches and systems. Section III introduces the Bi-temporal Timeline Index, the index maintenance and the query processing. Section IV gives details on the implementation of the temporal operators. Our approach is evaluated experimentally in Section V showing the high performance and low maintenance cost of the index. Section VI concludes the paper and provides some insights into future work.

II. Related Work

Storage methods for temporal data have been studied for several decades now and were described in a number of surveys in the late 1990s [8], [14]. Out of this large set, we cover methods that specifically provide indexes for a single temporal dimension as well as bi-temporal indexes. We first survey current commercial database systems.

A. Commercial Database Systems

Several commercial database systems have recently begun adding bi-temporal features, driven by – but not necessarily fully supporting – the SQL:2011 standard. Yet, as confirmed by the publicly available documentation and our recent analysis [6], the implementations are at an early stage, building on standard database storage and query processing and therefore achieving only limited performance: Teradata implements the Temporal Statement Modifier approach presented in [15] by Böhlen et al., which describes an extension of an existing query language with temporal features. However, the implementation is entirely based on query rewrites [3] which convert a bi-temporal query into a semantically equivalent non-temporal counterpart. IBM DB2 [2], Oracle as well as the production version of SAP HANA use a fixed horizontal partitioning between tuples that are currently valid in system time and those that have been invalidated in the past. DB2 and HANA perform all temporal operations directly on these tables, while Oracle uses a background process to move invalidated tuples from the undo to the Flashback Data Archive [16]. None of these systems has any specialized temporal indexes. In the production version SAP HANA only a limited set of temporal operators on system time are implemented.

B. Indexes for a single time dimension

The majority of research focused on indexing a single time dimension, either application or system time. Generally speaking, there are two main classes of index data structures for temporal data: 1) tree structures and 2) log sequences.

Given their general availability and maturity, B-trees are a promising basis for temporal indexes, yet their limitation on totally ordered domains for keys poses a significant challenge. Therefore, a large number of approaches to organize the keys for temporal data has been proposed, many stemming from interval storage: Time points for the boundaries of intervals [17] or composites of values and time (e.g., MAP21 [18]).

R-trees [19] were originally designed to index spatial data, but can naturally be used to store (time) intervals or combinations of keys and time. Some R-tree variants are optimized to meet the requirements specific for temporal indexing: The Historical R-tree [20] maintains an R-tree for each timestamp to efficiently answer time point queries.

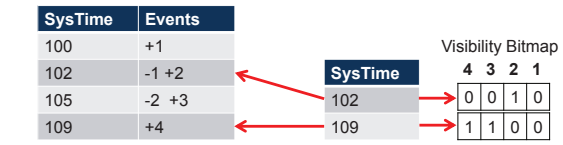
Multi-version techniques can be applied where trees are built for different versions in time such as the multi-version B-tree (MVBT) [11] and the multi-version 3D R-tree [21]. In principle, it is feasible to use a single dimension data structure to index the full state of the application time for the current system time. However, many data structures such as MVBT [11] exploit the append-only semantics of the system time and therefore cannot be applied for the application time.

C. Bi-temporal Indexes

Significantly less research has been done so far for indexing bi-temporal data. One straightforward way to index

	Name	City	Balance	SysStart	SysEnd
1	John	Smallville	$50	100	102
2	John	Largevill	$40	102	105
3	John	Largevill	$30	105	∞
4	Max	Newtown	$80	109	∞

Fig. 1: Temporal Customer Table



(a) System Time Event Map (b) Checkpoints

Fig. 2: Timeline Index for the Customer Table

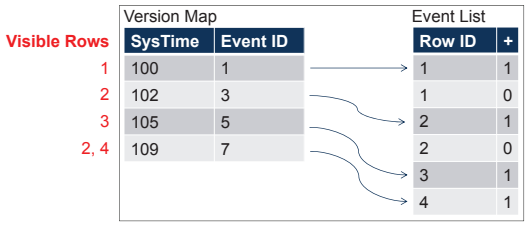


Fig. 3: Event Map Memory Layout

bi-temporal data is applying a spatial index structure over rectangles which are bounded by application and system time intervals. Such spatial indexes include the TP-Index [22], the GR-tree [23] and the 4R-tree [24]. Whereas this approach is very intuitive and most useful for simple selections, it does not allow for exploiting individual temporal orders and more complex temporal operations.

An approach to compensate for this issue is to decouple the application time and system time dimension. In theory, any two unitemporal index structures introduced in Section II-B can be combined to support bi-temporal indexing. A highly refined variant is the Multiple Incremental Valid Time Tree (M-IVTT) [10]. The M-IVTT follows a pattern of two-level bi-temporal indexing trees (2LBIT) [25], which use a B+-tree to index system time at the top level, whereas each leaf contains a pointer to an application/valid time tree (VTT) for each point in system time. This concept can further be improved by utilizing partial persistence [8], which takes into account that only tuples at the latest system time can be updated, whereas older versions are read-only. The bi-temporal interval tree (BIT) and bi-temporal R-tree (BRT) introduced in [8] exploit this partial-persistence methodology. The Bib+-tree [9] replaces the R*-tree in BRT with a R+-tree and manages the application time dimension based on a R*-tree.

In summary, there are only a few dedicated indexes for bi-temporal data which –like their one-dimensional counterparts– often do not exploit the properties of modern hardware.

D. System Timeline Index

The Timeline Index [13] is a data structure for the system time dimension. As it is the closest match to the Bi-temporal Timeline Index, we describe it in a separate section.

1) Index Data Structure: Figure 1 shows an example of a customer table that contains system time information. In this table the visibility intervals of temporal data is represented in a pair of additional columns *SysStart* and *SysEnd*. The idea of the Timeline Index is to keep track of all the visible rows of the temporal table at every point in time. Figure 2a illustrates this idea: For each system time where changes (i.e., Events) were applied to the table, an entry is appended to the *Event Map*: For inserted rows (called *activation*) we record the *Row ID* and for deleted rows (called *invalidation*) we use the negated Row ID. Updates are implemented by a deletion followed by an insertion. In the example of Figure 2, we see that at system time 102 the row for customer John was updated, i.e., row 1 was deleted and row 2 was inserted.

By scanning this index, operators can determine the changes between versions as well as compute the set of active tuples for a specific version. For example, to keep track of the set of visible rows at every point in time, we can maintain a bit vector (called *Visibility Bitmap*) for which each element indicates that the row is visible (bit is set) or not. As the temporal table may grow large, we would like to avoid the full scan of the Event Map. Therefore, the idea is to materialize the Visibility Bitmap generated during the full scan for specific versions (i.e., system time). We call such a materialized bit vector a *Checkpoint*. As shown in Figure 2b, a Checkpoint includes a Visibility Bitmap which represents the visible rows of the temporal table at a certain version. By controlling the number of Checkpoints, an administrator can perform a tradeoff between query cost and storage overhead. As we show in [13], the space overhead of this index is linear with very small constants.

2) Implementation: Figure 3 provides implementation details of the Timeline Index, which is optimized for scan-oriented access patterns, which are favorable for modern hardware. The *Event Map* is implemented by two main components: 1) The *Version Map* keeps track of the sequence of events produced by database transactions by mapping a system time to a set of events generated at a certain time. 2) The *Event List* is a chronological list of events, where each event is represented by a Row ID and the indicator for activation (1) and invalidation (0). The reference from the Version Map to the Event List is represented by the accumulated number of events that happened before a certain point in time. This design decouples the storage of the temporal table from the temporal order, so that the values in the temporal tables can be stored in any order, enabling better compression and partitioning. On the left-hand side of Figure 3 the visible rows for each system time are shown in red. The two data structures are append-only, i.e., once an entry has been inserted into the *Version Map* or *Event List*, none of its fields will ever be updated again. This restriction is sufficient for indexing system time but not acceptable for application time (see Section III).

3) Index Construction: The index maintenance algorithms have linear complexity with respect to the number of events, since every tuple needs to be touched exactly twice. In addition, the Checkpoints can be generated during the index construction. For efficient look-up of the relevant Checkpoint,

	Name	City	Balance	StartApp	EndApp	StartSys	EndSys
1	John	Smallville	50	10	∞	100	102
2	John	Smallville	50	10	11	102	∞
3	John	Largevill	40	11	∞	102	105
4	John	Largevill	30	11	13	105	110
5	John	Costtown	100	13	14	105	110
6	John	Largevill	30	14	∞	105	106
7	John	Largevill	30	14	16	106	110
8	Max	Newtown	80	15	∞	109	∞

Fig. 4: Bi-temporal Table

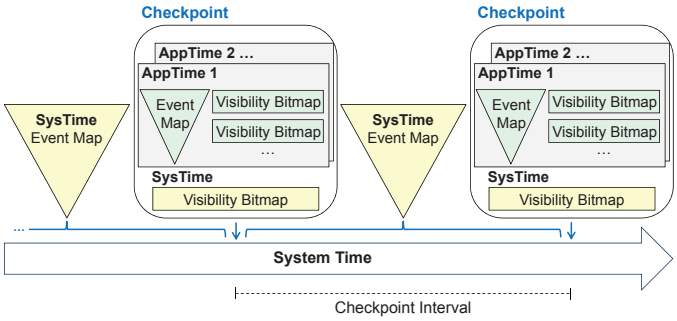


Fig. 5: Bi-temporal Timeline Index Architecture

the system time is stored at which the Checkpoint was taken together with the position within the Event List. New Checkpoints can be computed incrementally.

III. Bi-temporal Timeline Index

In this section we introduce the *Bi-temporal Timeline Index* which generalizes the Timeline Index towards the full bi-temporal data model of the SQL:2011 standard.

As outlined in Section I, the key design driver is the observation that the majority of bi-temporal queries are dominated by one dimension, ignoring the others or constraining them to a single point in time. Typical examples are complex temporal analytics over application time at the current system time or some specific past system time. We therefore prefer to use dedicated single-dimension temporal indexes over multi-dimensional indexes. Updates to application time indexes may change the application time past, but can only be performed on the most recent application time index, simplifying the update requirements over a pure application time index. Finally, we will not store application indexes for all system time points to minimize storage requirements.

Similar design principles have also been applied for the M-IVTT [10], but the log-based design of Timeline yields a much simpler design with lower space utilization, cheaper index maintenance cost and higher query performance.

A. Index Data Structure

Figure 4 shows the example data used in Figure 1, but now extended with a single application time dimension, referred to as *StartApp* and *EndApp*. In this example, the application time refers to the time when people actually lived in a certain city, whereas the system time (denoted as *StartSys, EndSys*) refers to the time when changes were recorded in the database. We

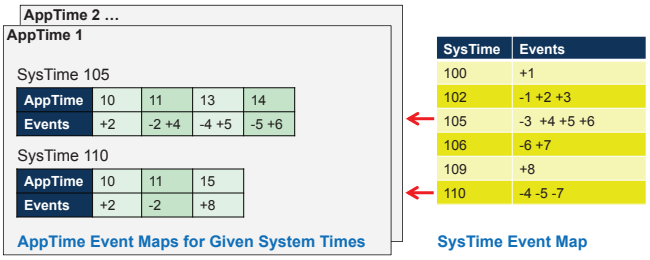


Fig. 6: Bi-temporal Timeline Index for the Table in Figure 4

will use this example to illustrate the additional complexity introduced by a bi-temporal workload.

First, updates of the application time require a new version of the database. That is, a modification in application time implies a new version in system time. The opposite is not necessarily true. Second, application time updates may change values that were considered "past", e.g., by changing the city where John lived from application time 10 to 11 at a later point in system time.

As shown in Figure 5, the Bi-temporal Timeline Index extends the Timeline Index by maintaining an application time *Event Map* and a set of *Visibility Bitmaps* for every application time dimension in every Checkpoint. This application time Event Map and Visibility Bitmaps can directly be used to slice (or join or aggregate) in application time, if the query matches the Checkpoint in system time. Otherwise, we need to consider the system time Event Map in order to pick up all events that may have changed the application time after the Checkpoint.

The Bi-temporal Timeline Index for our running example is given in Figure 6 (for simplicity we omit application time Visibility Bitmaps). For instance, to find out where John lived at application time 11 according to the state of the database at system time 105, we consult the application time Event Map denoted "SysTime 105" in the top left corner of Figure 6. The application time Event Map tells us that row 4 is visible for application time 11. The concrete change is only stored in the table, i.e., that John lived in Largevill.

A single Bi-temporal Timeline Index is sufficient for each temporal table. The frequency of Checkpoints and the choice for which application time dimensions to create an application time Event Map is tunable based on the workload.

B. Index Construction

The construction of a Bi-temporal Timeline Index is similar as for system time only. Yet, for each application time dimension, at each Checkpoint we create an (application time) Event Map and a set of Visibility Bitmaps, considering all tuples that are visible at the system time of this Checkpoint.

Again, we build the index incrementally starting from a previous Checkpoint in order to limit the scope of this scan through the temporal table. The process of how to construct a new (updated) application time Event Map from a previous Checkpoint incrementally is depicted in Figure 7, which shows the changes to the underlying data, either as additions (tuple 8) or as system time invalidations (EndSys of tuples 6-7). The starting point is the application time Event Map from the

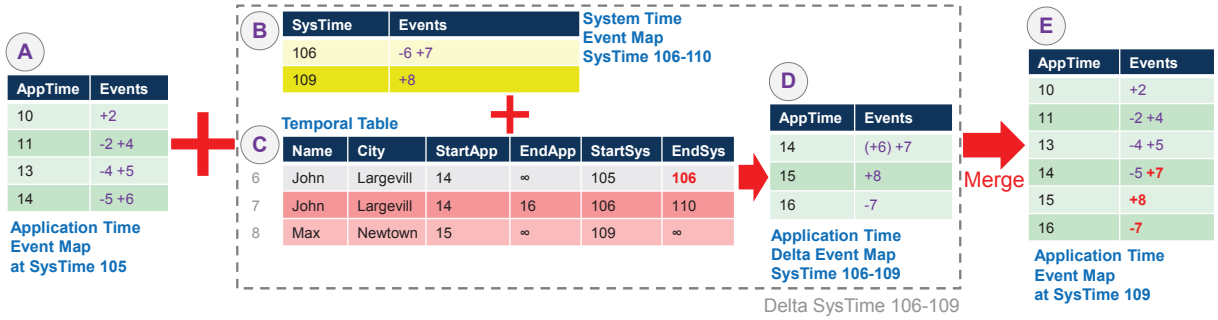


Fig. 7: Incremental Construction of one Application Time Event Map

previous Checkpoint, taken at system time 105 in this example and denoted as **(A)** in Figure 7. Furthermore, we compute a *Delta* (denoted as **(D)**) against this application Event Map using the system time Event Map **(B)** and the temporal table **(C)**. This Delta contains insertions and deletions (denoted as "()") of events that occurred after the Checkpoint (i.e., from system times 106 to 109 in this example). For instance, at system time 109, Tuple 8 is added, which affects an event at application time 15 so that the Delta records a "+8" event at application time 15. As another example, the deletion of Tuple 6 at system time 106, invokes a *deletion* event in the Delta: In order to express that Tuple 6 should be removed from the Event Map, we encode this *deletion* event as "(-6)".

As a final step to construct the new application time Event Map at system time 109, the Event Map from system time 105 **(A)** is merged with the Delta **(D)**: insertions in **(D)** (e.g., +8 at Time 15) are added to **(A)**; invalidations in **(D)** (e.g., -7 at Time 16) are also added to **(A)**; deletions in **(D)** (e.g., (+6) at Time 14) result in deleting the entries from **(A)**. A linear merge is performed, as **(A)** and **(D)** are sorted equally.

Once an application time Event Map has been created it is immutable because it is valid for a fixed version in system time. As a result, the index can be stored in a read-optimized form. Creating the delta explicitly instead of applying the changes directly allows us to decouple the application time index computation and cache these deltas for later use.

Given the space constraints of the paper and the (relative) simplicity of the Timeline Index, we will provide just a sketch of the time and space complexity of the data structure and its maintenance operations. Without Visibility Bitmaps, the space complexity is $\mathcal{O}(k*N)$, where k is the number of checkpoints and N the size of the temporal table, as an application time Event Map may contain all events in the worst case. Visibility Bitmaps increase the cost to $\mathcal{O}(k^2*N)$, since each Visibility Bitmap contains N bits and we have k bitmaps for each application time and for the system time dimension. The index is indeed linear to the number of events, and the quadratic impact of checkpoints is typically offset by (1) their small number, (2) the small constants of bitmap sets with additional compression potential and (3) the fact that users can control the overheads as a tradeoff between storage space and query response times. We will investigate the general time-space tradeoff as part of the experiments in Section V-H. Creating deltas and merging them is linear to the number of events involved, again possibly dominated by the cost accessing every element in the Visibility Bitmaps.

IV. Index Usage

In this section, we explain how the Bi-temporal Timeline Index supports a wide range of access patterns and operators.

A. Index Access Patterns

The Bi-temporal Timeline Index supports temporal queries on system and multiple application time dimensions. In this section, we will first describe all combinations of system and a single application time and then consider multiple application times. We express all these access patterns as range queries on a temporal range $[s, e]$, where $\perp$ denotes an unspecified point in time. A query may access each time dimension in 3 different ways, generalizing the concepts of current, sequenced and non-sequenced, as defined by Snodgrass [26]:

- **Point in Time** $[s, s]$. All tuples are selected which are visible at a particular point in time s.
- **Range** $[s, e]$. A range $[s, e], s < e$ means we look at a (closed) time interval. All tuples are added to the result whose visibility interval overlaps.
- **Agnostic** $[\perp,\perp]$. There is no restriction for this time domain, all tuples are selected.

Table I gives an overview how we can use the Bi-temporal Timeline Index for different combinations of these access patterns. Let us consider the case where both dimensions are constrained to a point ($[Ts, Ts]/[Ta, Ta]$), which may be used for timeslice in both dimensions. We already used this case as an example in Section III-A by showing how the Bi-temporal Timeline Index of Figure 6 can be used to find out where John lived at application time 13 for a database at system time 108.

We start from latest previous Checkpoint (i.e., at system time 105 in this example), which gives us access to 1) the set of all tuples that are active at that system time and 2) an application time Event Map and Visibility Bitmaps at this point. We search for the nearest previous (application time) Visibility Bitmap, which provides us with the information on the tuples that are active in application time. We then traverse the application time Event Map to retrieve the event in the application time domain until we reach the desired point in application time. If the requested system time corresponds to the system time of the Checkpoint, we are done. If not, we

SysTime	AppTime	Index Usage
$[Ts, Ts]$	$[Ta, Ta]$	• Search latest previous Checkpoint based on SysTime Ts • Search latest previous AppTime Visibility Bitmap for Ta • Follow AppTime Event Map until Ta is reached (toggle bits) • Follow SysTime Event Map until Ts is reached (toggle bits, apply events only for tuples visible for Ta)
$[Ts, Ts]$	$[Ta, Tb]$	• Like $[Ts, Ts]/[Ta, Ta]$, but continue following AppTime Event Map until Tb (set bits to true for all activated tuples in $[Ta, Tb]$ to implement a union operation) • Follow SysTime Event Map until Ts is reached (toggle bits, apply events only for tuples visible for $[Ta, Tb]$)
$[Ts, Ts]$	$[\bot, \bot]$	• Like $[Ts, Ts]/[Ta, Ta]$, but only use System Timeline Index
$[Ts, Tt]$	$[Ta, Ta]$	• Like $[Ts, Ts]/[Ta, Ta]$, but continue following Event Map until Tt • Set bits for all activated tuples in $[Ts, Tt]$
$[Ts, Tt]$	$[Ta, Tb]$	• Like $[Ts, Ts]/[Ta, Tb]$, but continue following Event Map until Tt is reached (set bits, apply events only for tuples visible for $[Ta, Tb]$)
$[Ts, Tt]$	$[\bot, \bot]$	• Ignore the application time
$[\bot, \bot]$	*	• Do a table scan instead because the Timeline Index would be inefficient

TABLE I: Index Usage for Different Access Patterns

use a simplified and more efficient variant of the technique described in Figure 7: the deltas from the system time Event Map are scanned until the requested point in system time, but not merged. Instead we directly filter the results, which saves the cost of building and merging the delta index.

On purpose, we do not target queries that have no constraints on system time. While building an application timeline over the whole system time would clearly be feasible, we have not encountered any use cases requiring such support – in particular since a table scan provides a convenient fallback.

Most operations can directly be executed in this way, as indicated by the other cases shown in Table I. Some operations, however, such as temporal aggregation or temporal join over application time only, require the presence of an application time Event Map at a specific system time. As we do not store a complete application time Event Map for each point in system time, we need to reconstruct the information from the Checkpoints and the system time delta at runtime. By changing the Checkpoint interval we can trade faster execution time for increased memory consumption. We consider three alternatives to retrieve the application time state for a given system time:

- **Recompute (R).** Rebuild the application time Event Map completely from scratch (not using checkpoints).
- **Index Delta Merge (M).** Retrieve the application time Event Map from the latest checkpoint, compute the application delta, merge both into a new Event Map.
- **Dual Index (D).** Retrieve the application time Event Map from the latest checkpoint, compute the application delta index and give both as an input to the temporal operators.

Recompute **(R)** is the slowest approach with a constant overhead, whereas the other two alternatives Delta Merge **(M)** and Dual Index **(D)** have similar performance. Given k checkpoints and the temporal table size N, the cost for **(R)** is $\mathcal{O}(N)$. The cost for creating the Delta is $\mathcal{O}(N/k)$ because the maximum size of the system time Event Map range is N/k. The Delta computation is required for **(M)** and **(D)**. The additional cost for the merge in **(M)** is $\mathcal{O}(N)$ as the maximum size of an application time Event Map in the checkpoint is $\mathcal{O}(N)$. Yet, the merging two indexes is much more efficient than rebuilding the index from the table.

(M) allows us to use the same implementation of our temporal operators for all time dimensions, whereas **(D)** requires an adapted operator implementation. We therefore use **(M)** for the experiments. For future work, we want to investigate if there are benefits from caching deltas as well as considering the next "future" checkpoint.

B. Bi-temporal Operators

Temporal Aggregation. Based on a temporal table aggregated values can be computed for groups of the timestamps of a tuple or, more generally, windows. Different variants of temporal aggregation have been described in literature [27]. In this paper, we will present the implementation of this operator by the example of *instantaneous* temporal aggregation. We also implemented other aggregation forms such as sliding window [28].

For a temporal aggregation over system time ($[Ts, Tt]/[\bot, \bot]$), indicated by `GROUP BY SYSTEM_TIME()` in our extended SQL syntax, we can immediately use the implementation of [13], relying on the system time Timeline Index: A linear scan over the Timeline Index (or the relevant range between checkpoints) yields the activations and deactivations of tuples, which can be used to incrementally compute the aggregate function. For a temporal aggregation over application time at a fixed point Ts in system time ($[Ts, Ts]/[Ta, Tb]$), using `GROUP BY APPLICATION_TIME()` in our syntax, we rely on an application time Event Map. If the chosen system time does not correspond to a checkpoint, we need to build the application time Event Map for Ts, which incurs the overhead described in Section IV-A. For both dimensions, the cost of index-based temporal aggregation is $\mathcal{O}(S)$, where S corresponds to number of events in for the aggregation range.

Timeslice. The timeslice operator retrieves those tuples that are visible for a given time T, i.e., the validity intervals of the tuples overlap T. In this paper we consider temporal conditions on both system and application time. Pure system time ($[Ts, Ts]/[\bot, \bot]$) can be computed efficiently by using only the system time Event Map and Visibility Bitmaps (see Table I, third row). A pure application time timeslice ($[\bot, \bot]/[Ta, Ta]$)

becomes a table scan, since there is no selection on the system time, and all application time Event Maps of the index are only valid for a specific point in system time. Constraining both dimensions ($[Ts, Ts]/[Ta, Ta]$) is implemented by first retrieving the tuples visible at Ts and post-filtering the tuples valid at Ta. Thus, we avoid the creation of an application time Event Map, as explained in Section IV-A.

The cost for a bi-temporal timeslice operator for k Checkpoints, the size of the system time Event Map M_S and application time Event Map M_A is $\mathcal{O}(2 \cdot \log(k) + M_A/k + M_S/k)$, stemming from the effort to locate the checkpoints and then range scan in the individual Event Maps.

Temporal Join. A temporal join returns all tuples from two temporal tables which satisfy a value predicate and whose time intervals overlap (i.e., they are valid at the same points in time). Our temporal join operator exploits the temporal order of both tables to perform a merge-join style temporal intersection, augmented by a hash-join style helper structure for the value comparisons. For the bi-temporal join we have to distinguish several cases for the join predicates: 1) If the temporal join predicate is only on system time (regardless on any non-join constraint on the application time), we rely on the system time Event Map and Visibility Bitmaps in available checkpoints to evaluate the join as described in [13]. 2) If the temporal join predicates use both system time and application time, we use the same algorithm as in 1) and apply the join predicate for application time after evaluating the predicate on system time. 3) Finally, if the temporal join predicate concerns only application time, we have two variants: (a) assuming temporal restrictions on system time for the inputs, we can construct the application time Event Map for the requested system times using the approach described in Section IV-A. (b) If the system time is unconstrained, we have to build a "global" application time event map. After this step, we can use the same algorithm as for 1).

Range Queries. A range query generalizes the definition of the timeslice operator to visibility intervals for one or many time dimensions. All tuples are included in the result for which the visibility interval overlaps.

As outlined in Table I, there is a wide range of options depending on ranges on each dimension. Whenever system time is involved, we get all visible tuples which are valid at the lower bound of the system time interval as described for the timeslice operator above. We then resume scanning the system time Event Map and apply the delta. Any other predicates including conditions on application time can be applied by accessing the temporal table for matching tuples.

Thus, the Bi-temporal Timeline Index supports application and system time effectively whenever the system time dimension is restricted. In case the index selectivity is too high, we have the option to fall back to a full table scan. In case of selecting the full system time range for a temporal join or aggregation on application time, it is also possible to rebuild an application Timeline Index at query execution time.

Data Set	SF_0	SF_H	|customer|	|partsupp|	|orders|	#versions
Tiny	0.01	0.1	0.2 Mio	0.08 Mio	0.4 Mio	0.1 Mio
Medium	1	10	7 Mio	8 Mio	9 Mio	10 Mio
Large	1	25	17 Mio	19 Mio	20 Mio	25 Mio

TABLE II: Data Set Properties

C. Dealing with Multiple Application Times

As described in Section III-A, one system time and multiple application time dimensions per table are supported by the Bi-temporal Timeline Index using Event Maps and Visibility Bitmaps for each dimension. As long as only a single application time is used in a query, the operators do not differ from their previous description besides chosing the right index. Likewise, if a query accesses multiple application time dimensions but does not correlate them, their actual computations work as before. We only need to adapt our operators if a single operator needs to deal with multiple application times. For *timeslice* or *range*, the story does not change much: the tradeoffs outlined in Table I are considered, and depending on availability and selectivity an additional index is used (with tuple ID intersection) or the predicate is evaluated as a filter. For *join*, compatible orders can be processed directly, leading to an n-way scan. If the orders do not fit, a more expensive tuple ID intersection or value filtering needs to performed. Finally, *temporal aggregation* relies on total order, so some kind of correlation (like a join) has to happen beforehand.

V. Experiments and Results

In this section we evaluate the performance of the Bi-temporal Timeline Index against several state-of-the-art index types as well as a commercial DBMS.

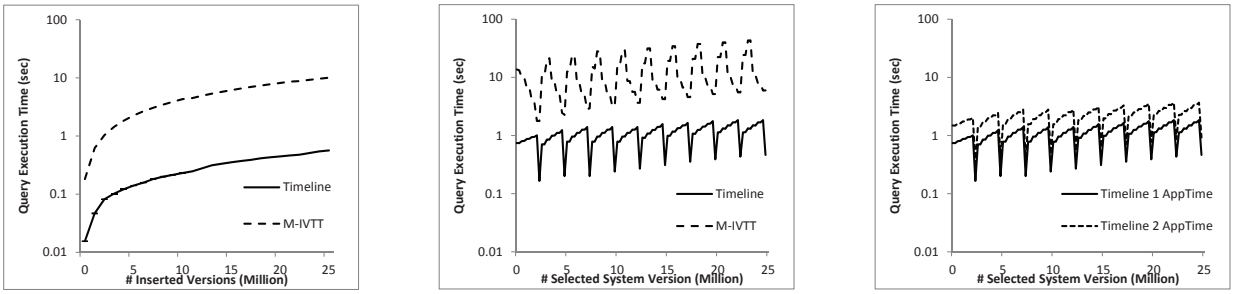
A. Software and Hardware Used

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. Our implementation of the Bi-temporal Timeline Index was integrated into an SAP-internal database prototype (used for feature staging) whose design closely matches the actual SAP HANA system. For all measurements we set a timeout of 60 minutes and repeated them 10 times after a warmup.

B. Benchmark

Benchmark Definition. In order to provide a good coverage of temporal workloads, we chose the data sets and selected queries from our TPC-BiH benchmark proposal [5]. The benchmark provides a bi-temporal schema, a data evolution workload produced by a generator and a set of queries stressing a comprehensive set of operators and access patterns.

Data Sets. The data generator from the TPC-BiH benchmark takes the output of the standard TPC-H generator as version 1 and adds a history to it by executing update scenarios (e.g., new order, deliver order, cancel order). These scenarios were designed to match real use-cases from SAP and its customers,



(a) System Time (b) One Application Time (c) Multiple Application Times

Fig. 8: Temporal Aggregation [Large Data Set]

providing a workload which corresponds to the properties of a real-life temporal database. Each update scenario results in one transaction that generates a new version in our temporal database. As shown in Table II, the size of the data set is determined by two scaling factors:

- SF_0: The scaling factor of the TPC-H generator.
- SF_H: The scaling factor determining the size of the history as number of update transactions (in Millions).

The schema of the TPC-BiH data set is based on the standard TPC-H schema, but it includes additional attributes reflecting the system time and application time dimensions.

We used three data sets as described in Table II:

- **Tiny Data Set**, for expensive/unoptimized operators.
- **Medium Data Set**, default workload with a short history.
- **Large Data Set**, extended history of Medium Data Set.

Systems. In the experiments we compared 4 competitors:

- Our Bi-temporal Timeline prototype (referred to as "**Timeline**") uses the data structures and algorithms introduced in this paper. Unless stated otherwise, we use 10 system Checkpoints and thus 10 application time Event Maps with 10 Visibility Bitmaps each.
- The **M-IVTT** [10] uses a two-level bi-temporal indexing tree to index bi-temporal data. As no source code was available from the authors, we developed our own implementation of M-IVTT. This is the best B-tree-based implementation for bi-temporal operators we are aware of. Similar to Timeline, we use 10 full VTTs.
- The **RR*-tree** is an optimized R*-tree reducing the imbalance caused by updates. For our experiments we used an RR*-tree implementation from the authors of [29]. It is the fastest R-tree-based version we know about.
- **System Y** is a commercial disk-based relational database with native support for bi-temporal features. Due to license regulations we are not allowed to reveal the actual name. We created indexes which have been recommended by the index advisor for each workload. We ensured that the entire workload was served from RAM after warmup.

C. Experiment 1: Temporal Aggregation

In the first set of queries we evaluate the performance of instantaneous temporal aggregation. This operator stresses the temporal order aspect significantly, as it traces the evolution of the data in the temporal dimension. As such, it is also a good representative for many temporal analyses such as window queries or time series. We utilized the TPC-BiH Query R.3b and varied the time dimension. We do not show any results for System Y, as the measurements already timed out for the *Tiny* workload. Likewise, no implementation of temporal aggregation is available for RR* at the moment, as it does not deliver the results in any temporal order. The following queries are evaluated for the *Large* data set.

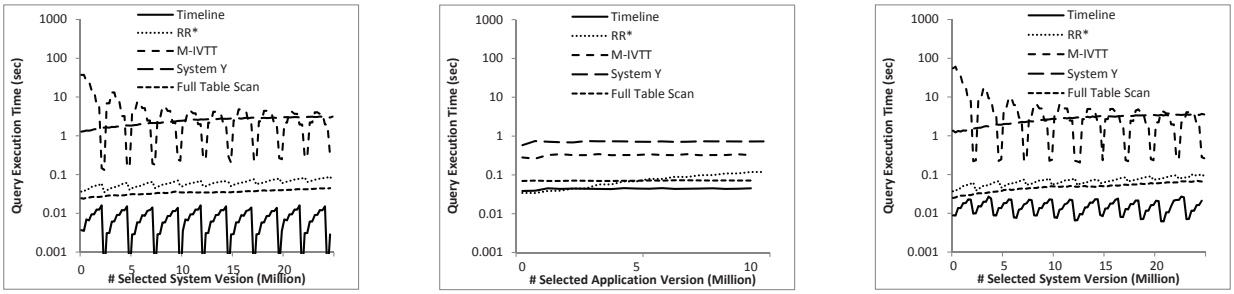
A1: Temporal Aggregation over System Time. We start our analysis with a temporal aggregation over system time for a fixed application time, using a selective aggregate function.

```
SELECT MAX(o_totalprice)
FROM orders o FOR
  APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'
WHERE o_orderstatus = 'O'
GROUP BY o.SYSTEM_TIME()
```

This query is evaluated for a variable history size, increasing the history in steps of 10% from 0 to the full data set for a fixed point in the middle of the application time range. As it is shown by Figure 8(a), the temporal aggregation algorithm based on the *Timeline Index* scales linearly with the size of the data set. The query execution time is about 1 second for a data set of 10 million versions, matching the results in [13]. This is expected, as the system time Timeline Index is scanned linearly for activations and invalidations of tuples, which can be exploited well for the computation of the aggregation.

On the other hand, *M-IVTT* also seems to scale linearly with the data set, but with a about an order of magnitude times slower execution time. The reason for the worse performance of M-IVTT are: 1) A large amount of time is spent on constructing a full snapshot of the valid time tree (VTT). 2) Scanning the VTT is less efficient than a scan of the Timeline Index because it results random access patterns by following pointers in the tree structure. 3) M-IVTT encodes the time interval as a single value, and thus, the encoding and decoding of an interval takes extra effort.

A2: Temporal Aggregation over Application Time. Here, the aggregation is performed over one application time:



(a) System Time Only (T1) (b) Application Time Only (T2) (c) Bi-temporal (T3)

Fig. 9: Timeslice [Large Data Set]

```
SELECT MAX(o_totalprice)
FROM orders FOR
  SYSTEM_TIME AS OF TIMESTAMP '[SYS_TIME]'
WHERE o_orderstatus = 'O'
GROUP BY ACTIVE_TIME()
```

We keep the size of the data set constant and vary the point in system time instead. For better visibility we show the version range from 0 to 5 million (out of a 10 million).

As Figure 8(b) shows, *Timeline* exhibits a sawtooth pattern with a generally flat trend. These variations in runtime correspond to the checkpoints on which application time Event Maps are kept. If such a Checkpoint/index is available, the aggregation is performed on the Timeline Index, leading to a dip in the graph. In turn, when no index is available we need to reconstruct the fitting Event Map from the existing index using the the Delta Merge **(M)** approach from the closest *previous* Checkpoint as outlined in Section IV-B.

M-IVTT is also able to perform a backwards scan from the next *following* Checkpoint, which is feasible but currently not implemented with Timeline. Therefore, M-IVTT produces a more symmetric pattern. Again, the performance of M-IVTT is an order of magnitude worse. This is due to the fact that in the case of application time the whole valid time tree has to be traversed as no patches are available for this time dimension.

A3: Temporal Aggregation over Multiple Application Times. In contrast to the previous query, we now consider multiple application time dimensions:

```
GROUP BY ACTIVE_TIME(), RECEIVABLE_TIME()
HAVING ACTIVE_TIME() = RECEIVABLE_TIME() + 10
```

For this temporal aggregation query, a group is created for point in time the set of tuples changes that are visible with respect to two application time dimension as of a fixed point in system time. We compare against Timeline for a single dimension and observe slightly less than twice the cost, since we perform a stepwise linear scan of two Timeline indexes. No other index structures are able to perform this query.

D. Experiment 2: Timeslice

The next (and most popular) class of queries covers the timeslice operator, which restores a certain state in time stressing the selection capabilities of the index. In bi-temporal settings, a timeslice operator can be applied to either dimension individually or on both. For the queries in this section we adopt TPC-BiH Query T.1 and vary the point in each time dimension, using `TEMPORAL_CONDITION` as a placeholder:

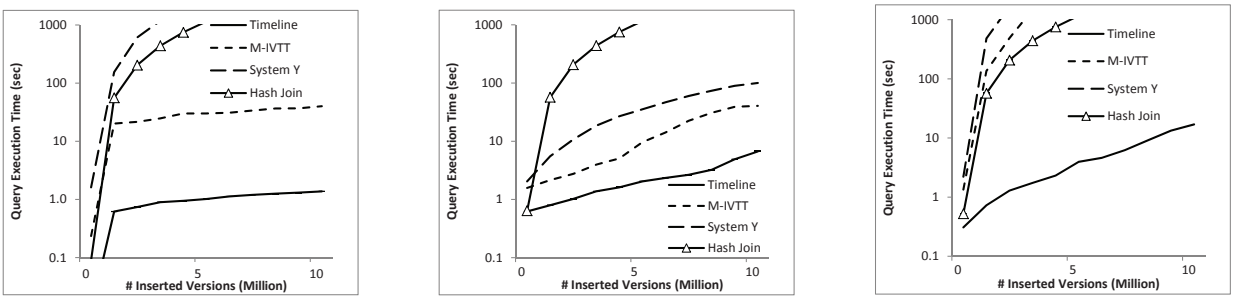
```
SELECT AVG(ps_supplycost)
FROM partsupp
TEMPORAL_CONDITION
```

The queries are measured on the *Large* data set, varying the selected version. Figure 9 summarizes our results. Since an application timeslice without a system time constraint is not supported by our index structure, we omit the results.

T1: Timeslice for System Time Only. The first query (Figure 9(a)) performs a timeslice to a given point in the system time dimension while considering the entire application time. Hence, we replace the placeholder `TEMPORAL_CONDITION` with `FOR SYSTEM_TIME AS OF TIMESTAMP '[SYS_TIME]'`.

For *Timeline* we see once more a sawtooth pattern as the evaluation starts on the closest previous Checkpoint on the system timeline, retrieves the bitmap containing the tuples valid at this point and traverses this index sequentially until it reaches the desired version. The performance is always clearly better than an in-memory table scan, typically by more than an order of magnitude faster. *M-IVTT* also shows the symmetric sawtooth pattern driven by the reconstruction of the VTTs, but the performance is significantly worse, at least an order magnitude worse than a table scan in the best case. *RR** performs better, since it can answer selection queries directly. Yet, the overhead of the tree index, including probes to overlapping regions, prevents it from outperforming the table scan. The query execution time of the commercial row-store *System Y* is almost three orders of magnitude slower. It always performs a full table scan, since the temporal filter is not selective enough to benefit from a conventional index.

T2: Timeslice for Both Time Dimensions: Vary App Time The next query performs a timeslice to a given point in application time for the *current* system time version, where we use `FOR APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'`. *Timeline* shows a constant performance, but is slower compared to T1. A timeslice to the *current* system time needs to be performed, and the result is filtered according



(a) Join on System Time (J1) (b) Join on Application Time (J2) (c) Join on Two Dimensions (J3)

Fig. 10: Temporal Join [Medium Data Set]

to the application time condition. Timeline therefore performs similar to a full table scan. *M-IVTT* needs to perform the same kind of VTT reconstruction for all points in application time and therefore shows a constant performance, but it again does not outperform a table scan. *RR** and *System Y* perform similarly as for T1. Given the result sizes, these systems perform slightly faster and see an increased runtime for higher application times as the size of the results grows.

T3: Timeslice for Both Time Dimensions: Vary Sys Time The final timeslice query keeps the application time fixed to a point in the middle of the time range, and the system time is varied on the x-axis. *Timeline* shows the familiar sawtooth pattern caused by the Checkpoints. Within a Checkpoint the application time Visibility Bitmaps and Event Maps are exploited to compute the application timeslice for the system time of the checkpoint. Next, the system time Event Map is applied for tuples matching in application time only, adding some additional cost compared to T1. *M-IVTT*, *RR** and *System Y* show very similar behavior as in T1 since the workload is influenced by the system time constraints.

The results for timeslice show that *Timeline* is very competitive, whereas the competitors do not outperform table scans.

E. Experiment 3: Temporal Join

The third class of experiments examines temporal joins. This operation retrieves all tuples from different tables whose validity intervals overlap, i.e., which are visible at the same time. As such it stresses the index structures for correlations, complementing the two previous experiments. We examine temporal joins with join conditions on (1) system time, (2) application time and (3) both time dimensions.

We utilize the following query, which is a non-temporal equijoin with a temporal join condition `TEMPORAL_CORRELATION` and a timeslice specification `TEMPORAL_CONDITION`: "Which expensive orders were open while the related customers had a low balance".

```
SELECT COUNT(*)
FROM customer c TEMPORAL JOIN orders o
  ON TEMPORAL_CORRELATION
TEMPORAL_CONDITION
WHERE c_custkey = o_custkey
  AND o_orderstatus = 'O'
  AND o_totalprice > 5000 AND c_acctbal < 100
```

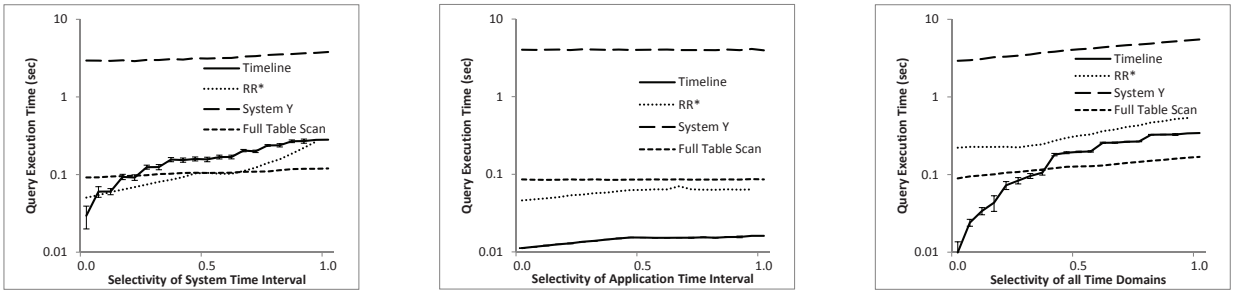
The results for this experiment are shown in Figure 10. The selectivities of the join predicates are depicted in Table III. The following queries are measured on the *Medium* data set.

J1: Temporal Join on System Time. The first experiment (depicted in Figure 10(a)) shows the results for performing a temporal join over the system time domain `ON c.SYSTEM_TIME OVERLAPS o.SYSTEM_TIME` and a fixed application time `FOR APPLICATION_TIME AS OF TIMESTAMP '[APP_TIME]'`. When changing the size of the history, *Timeline* scales linearly with the number of versions since it performs a concurrent scan over both indexes, efficiently merges the time-ordered lifetime intervals and directly evaluates the value join predicate, as outlined in Section IV-B. This way, the set of join candidates can be bounded effectively. *M-IVTT* can use the same algorithm, but needs to pay much higher index access cost. *System Y* cannot exploit any of the temporal semantics and is slowed down by the combinatorial explosion of versions. *RR** is even worse, since the spatial join algorithms provided with it only consider temporal overlap but not value correlations, leading to timeout even in the *Tiny* workload. We measured a standard *Hash Join* to investigate a join algorithm on the value domain. Similar to System Y, it is not effective, as it only exploits the value domain.

J2: Temporal Join on Application Time. In turn, the second experiment (depicted in Figure 10(b)) shows the results when we perform a temporal join over the application time domain `ON c.APPLICATION_TIME OVERLAPS o.APPLICATION_TIME` and fix the system time by `FOR SYSTEM_TIME AS OF TIMESTAMP '[SYSTEM_TIME]'`, mirroring the workload of J1. *Timeline* fares slightly worse since it needs to pay the cost of reconstruction an application time Event Map for the particular system time. Given the different index organization, *M-IVTT* has now lower delta reconstruction cost but is still more expensive than Timeline. *RR** and *Hash Join* fare roughly the same way as in J1, while *System Y* benefits from a lower temporal selectivity.

Join Query	Foreign Key	Time Domains	Combined
J1	0.013%	78.8%	0.012%
J2	0.013%	99.6%	0.013%
J3	0.013%	78.5%	0.012%

TABLE III: Join Selectivities on the Filtered Tables



(a) Full App.Time, vary Sys.Time (R1) (b) Fixed Sys.Time, vary App.Time (R4) (c) Vary Sys.Time, vary App.Time (R5)

Fig. 11: Range Queries [Large Data Set]

J3: Temporal Join on System and Application Time. Our last experiment for temporal joins (Figure 10(c)) correlates on both time dimensions and thus drops the timeslice present in the previous experiments, making it the most demanding workload due to further combinatorial effort. As a result, all approaches see further cost increases. Yet, *Timeline* still copes best, as it is able to exploit both time and value constraints.

We also performed experiments where we varied the ratio between temporal and spatial selectivity, which we omit for space reasons. The experiments showed that the hybrid time/value approach for temporal joins supported by a time-ordered index works best. Although being (rarely) outperformed by other methods at extreme selectivities, it always comes close to the best approach and usually beats its competitors.

F. Experiment 4: Range Queries

Our last query performance experiment explores how well *Timeline* deals with arbitrary range selections. Given that it decouples the time dimensions, we may expect it to perform worse for arbitrary selections than a dedicated spatial index such as RR*. For this experiment we investigate the following query pattern, which includes two time dimensions.

```
SELECT COUNT(*), AVG(ps_supplycost)
FROM partsupp
FOR APPLICATION_TIME BETWEEN
  '[APP_TIME_LOWER]' AND '[APP_TIME_UPPER]'
FOR SYSTEM_TIME BETWEEN
  '[SYS_TIME_LOWER]' AND '[SYS_TIME_UPPER]'
```

The queries are evaluated for the *Large* data set. We examine 5 parameter settings: full range in one dimension, varying the other (leading to two experiments), fixing one dimension and varying the other (again leading to two experiments) and finally varying both dimension in concert. Selected results are shown in Figure 11, and we will discuss all of them here: Figure 11(a)/R1 yields the full application time and varies the size of the system time interval by decreasing the lower interval bound `[SYS_TIME_LOWER]`. Given the high selectivity of this workload, table scans are only outperformed for small system time intervals. *Timeline* holds up rather well against *RR**, even beating it at low selectivities. R2 (not shown) inverts this workload by taking the full system time range and varying application time. Given its design *Timeline* cannot directly support this query, and we rely on scans. R3 (not shown) fixes the application time to a point and varies the system time, leading to results like to R1. Figure 11(b)/R4 fixes the system time and varies the application time range, allowing *Timeline* to outperform all competitors. Finally, in Figure 11(c)/R5, we change both time ranges simultaneously. *Timeline* scales well, as it benefits from its system time index.

In summary, Timeline supports temporal range queries efficiently: Timeline provides performance similar to dedicated indexes and outperforms full table scans in most cases.

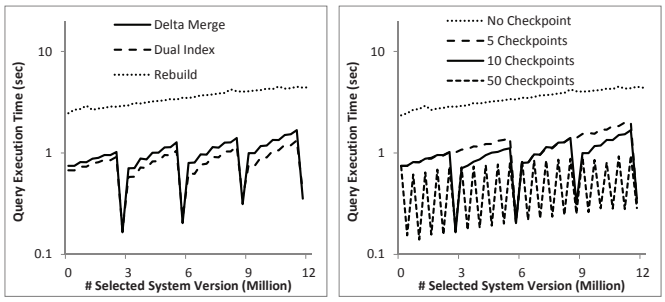
G. Experiment 5: Index Creation Time

One of the key goals of Timeline is the ability to quickly create indexes when needed, in particular for two scenarios: 1) Building an index from scratch when loading data and 2) Creating the appropriate application time index. Table IV shows the time of the index creation for the PARTSUPP table and different sizes of the data set. As it can clearly be seen, *Timeline* is the only index structure that scaled almost linearly and is fast enough to allow ad-hoc index creation for almost all workloads. In contrast to *M-IVTT* and *RR**, it only requires two scans instead of sorting or tree operations. The cost for a creating a bi-temporal Timeline Index are about 3 times higher than *Timeline SysTime*, which indexes system time only.

The tradeoffs on generating intermediary application time Event Maps (as outlined in Section IV-A) are more diverse: Figure 12(a) compares alternative reconstruction approaches for temporal aggregation. Building a Timeline Index from scratch is always slower than incorporating existing snapshots. Merging the Event Map with the changes (*Delta Merge*) is often slightly outperformed by running adapted operators on the snapshot and the changes separately (*Dual Index*), but allows us to keep the complexity of operators low. Given that the benefits of Dual Index are limited, we focused on Delta Merge in this evaluation. The difference between dips and peaks of around 1.5 seconds indicates the maximum cost of delta construction and merge. Comparing this value with the results of direct evaluations for timeslice in Figure 9(c) confirms our decision of only reconstructing an Event Map when needed, as the cost of scanning is around 0.2 seconds.

Data Set	Timeline	Timeline Sys	M-IVTT	RR*	System Y
Tiny	0.9	0.3	1.6	0.25	1.8
Medium	3.8	1.1	268.9	33.9	32.2
Large	7.8	2.7	504.7	85.0	128.8

TABLE IV: Index Construction Time (sec)



(a) Alternative Index Construction (b) Variable Checkpoints Number

Fig. 12: Timeline Index Property Settings [Large Data Set]

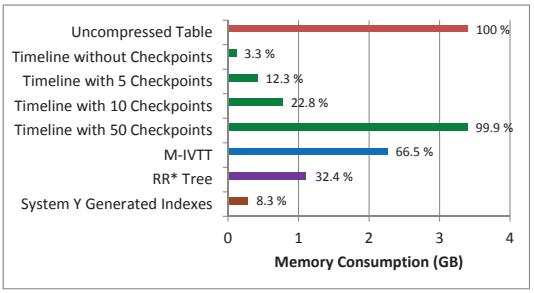


Fig. 13: Memory of PARTSUPP Table [Large Data Set]

H. Experiment 6: Memory Consumption

In order to visualize the time-space tradeoff, Figure 12(b) shows the effect of variable checkpoint intervals by the example of the temporal aggregation operator. In addition, Figure 13 shows the memory consumption for each index data structure when loading the PARTSUPP table of the *Large* data set. As a good compromise, we chose 10 Checkpoints and 10 Visibility Bitmaps for application time per Checkpoint as well as 10 VTTs for M-IVTT for our experiments. The cost for *Timeline* is dominated by the number of Checkpoints: without Checkpoints it only requires around 3% of the space of the temporal table. Checkpoints drive up this cost – in our case with 10 Checkpoints we end up at 23% of the temporal table. Despite never outperforming Timeline, *M-IVTT* requires significantly more storage. Likewise, *RR** is more expensive than a Timeline Index with Checkpoints. The story for *System Y* is quite complex due to the results of the index advisor: The indexes for the *Large* PARTSUPP require only around 8%, but they support the timeslice operator only. Furthermore, slight workload variations can lead to drastic changes in indexing, e.g., PARTSUPP for Medium triggers additional indexes due to different selectivities, requiring 51%.

VI. Conclusion

In this paper we proposed an index for bi-temporal data that exploits for properties of modern hardware such as large main memory and fast scans. The key idea is that the individual order in each dimension is more relevant than the (asymmetric) spatial properties of the two time dimensions. As a result, we use dedicated one-dimensional index structures for each domain, where the application index is only materialized at specific checkpoints. Computing an intermediate index is fast, and as a result the operations such as selection, joins and temporal aggregation on this index outperform state-of-the-art implementations of temporal and spatial index structures by orders of magnitude.

As temporal tables can become quite large, it may not always be feasible to keep all temporal data in the main memory of a single machine. Consequently, we are investigating how temporal tables and the corresponding index structures can be partitioned onto a cluster. We also plan to evaluate the Timeline Index for alternative storage such as disk and flash.

References

[1] M. Doane, *The SAP Blue Book - A Concise Business Guide to the World of SAP*. SAP Press, 2012.

[2] C. M. Saracco et al., "A Matter of Time: Temporal Data Management in DB2 10," IBM, Tech. Rep., 2012.

[3] M. Al-Kateb et al., "Temporal Query Processing in Teradata," in *EDBT*, 2013.

[4] K. G. Kulkarni and J.-E. Michels, "Temporal Features in SQL: 2011," *SIGMOD Record*, vol. 41, no. 3, 2012.

[5] M. Kaufmann et al., "TPC-BiH: A Benchmark for Bi-Temporal Databases," in *TPCTC*, 2013.

[6] M. Kaufmann et al., "Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?" in *EDBT*, 2014.

[7] A. Dignös, M. H. Böhlen, and J. Gamper, "Temporal alignment," in *SIGMOD*, 2012.

[8] A. Kumar et al., "Designing Access Methods for Bitemporal Databases," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 1, 1998.

[9] Q. L. Le and T. K. Dang, "Bib+-tree: An efficient multiversion access method for bitemporal databases," in *iiWAS*, 2009.

[10] M. A. Nascimento et al., "M-IVTT: An Index for Bitemporal Databases," in *DEXA*, 1996.

[11] B. Becker et al., "An Asymptotically Optimal Multiversion B-Tree," *VLDB J.*, vol. 5, no. 4, 1996.

[12] H. Plattner, "A Common Database Approach for OLTP and OLAP using an In-Memory Column Database," in *SIGMOD*, 2009.

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

[14] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data," *ACM Comput. Surv.*, vol. 31, no. 2, Jun. 1999.

[15] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal statement modifiers," *ACM Trans. Database Syst.*, vol. 25, no. 4, 2000.

[16] R. Rajamani, "Oracle Total Recall / Flashback Data Archive," Oracle, Tech. Rep., 2007.

[17] H. Edelsbrunner, "A New Approach to Rectangle Intersections Part I," *International Journal of Computer Mathematics*, vol. 13, no. 3-4, 1983.

[18] M. A. Nascimento and M. H. Dunham, "Indexing Valid Time Databases via B+-Trees," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 6, 1999.

[19] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD*, 1984.

[20] Y. Tao and D. Papadias, "Efficient Historical R-Trees," in *SSDBM*, 2001.

[21] Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," in *VLDB*, 2001.

[22] H. Shen et al., "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases," in *ICDE*, 1994.

[23] B. Rasa et al., "R-Tree Based Indexing of Now-Relative Bitemporal Data," in *VLDB*, 1998.

[24] B. Rasa et al., "Light-Weight Indexing of General Bitemporal Data," in *SSDBM*, 2000.

[25] M. A. Nascimento et al., "Efficient Indexing of Temporal Databases via B+-Trees," 1996.

[26] R. T. Snodgrass, *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[27] J. Gamper, M. H. Böhlen, and C. S. Jensen, "Temporal aggregation," in *Encyclopedia of Database Systems*, 2009, pp. 2924–2929.

[28] M. Kaufmann et al., "Comprehensive and interactive temporal query processing with SAP HANA," *PVLDB*, vol. 6, no. 12, 2013.

[29] N. Beckmann and B. Seeger, "A Revised R*-Tree in Comparison with Related Index Structures," in *SIGMOD*, 2009.