Indexing Bi-temporal Windows

Chang Ge [†], Martin Kaufmann [#§], Lukasz Golab [†], Peter M. Fischer [*], Anil K. Goel [§]

[†]University of Waterloo
Waterloo, Ontario, Canada
{c4ge,lgolab} @uwaterloo.ca

[#]Systems Group
ETH Zurich, Switzerland
martinka@inf.ethz.ch

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

[§]SAP AG
Walldorf, Germany
anil.goel@sap.com

ABSTRACT

Bi-temporal databases support system (transaction) and application time, enabling users to query the history as recorded today and as it was known in the past. In this paper, we study windows over both system and application time, i.e., *bi-temporal windows*. We propose a two-dimensional index that supports one-time and continuous queries over fixed and sliding bi-temporal windows, covering static and streaming data. We demonstrate the advantages of the proposed index compared to the state-of-the-art in terms of query performance, index update overhead and space footprint.

1. INTRODUCTION

Bi-temporal databases support applications such as financial reporting, scientific data management, and data auditing, keeping track of changes to information over time (application time) and the time when they were recorded in the database (system time). This enables historical queries, such as "What were the total sales last November?", and roll-back queries against prior versions of the data, such as "What were the total sales last November as recorded last December?". Basic bi-temporal features are part of the recent SQL:2011 standard [17] and are being added to relational DBMSs.

Temporal attributes naturally lead to window queries, which are important building blocks of complex analytics. Windows may be *fixed* ("What were the total sales last November?") or *sliding* ("Every day, compute the total sales over the past 30 days"). There has been a great deal of work on sliding windows, especially in data stream processing [11], where windows provide a finite view of unbounded data. However, that work generally considers a single time dimension, and there is no universally agreed-upon way of treating time. Some systems construct windows according to tuple arrival, expressing system time. Other systems consider application timestamps, but assume that tuples arrive nearly in order. However, in bi-temporal databases, both time dimensions matter and may be uncorrelated: for example, information about dinosaurs may have been added recently but the application times may be millions of years in the past.

SSDBM '15, June 29 - July 01, 2015, La Jolla, CA, USA
© 2015 ACM. ISBN 978-1-4503-3709-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2791347.2791373

1.1 Motivating Examples

Consider a bi-temporal database storing information about customers and their accounts. A transaction, occurring at some system time, updates the account balance of one or more customers as of some application time. For example, on June 1, the database may be updated with the fact that John's balance between April 1 and April 30 was 1000. On August 1, John's balance between April 1 and April 30 may be updated to 500; perhaps the initial balance was incorrectly computed or there was a billing dispute that was settled on August 1. Both versions of John's balance will be stored with the corresponding system and application times. The first version has a system start time of June 1 and a system end time of July 31, whereas the second version has a system start time of August 1 and an unbounded system end time. Both versions have application start times of April 1 and application end times of April 30.

A fixed bi-temporal window query may be "Return the account balances for the application time period of April 1 to 30 as they were known between (system) times May 1 and May 30". We can also leave one window fixed and slide the other. If the application time window is sliding, we may ask "From January 1 till December 1, return the customer balances for the past 30 days, as they were known between December 1 and 30". If the system time window slides, we may ask "At the end of each day, return the customer balances for the application time period of April 1 to 30 as they were known in the past 14 days". Finally, both system and application windows may slide, e.g., "Return the customer balances for the past 30 days as they were known over the past 14 days".

Bi-temporal windows identify fragments of history—or the future in case of forecasting databases, which store predictions as tuples with future application timestamps—as it was known now or in the past. This enables historical analysis over different versions of the data, which is useful in financial reporting, inventory management, processing scientific or medical experiments, auditing and data cleaning. In the above examples, bi-temporal window queries support what-if scenarios, re-create and track account projections as of different times in the past, and characterize the impact of data modifications on financial reports generated at different times.

In addition to historical analysis, *continuous* bi-temporal queries can be supported by defining a system time window over newly arrived data and an arbitrary application time window. Such queries generalize continuous queries in a single time dimension, and show how our knowledge of the past (or the future) is evolving as new data arrive. Such queries naturally appear in Data Stream Warehouses [10], which support unified processing of streaming and historical data. The append-only nature of updates in bi-temporal data matches well with continuous query processing models.

1.2 Challenges and Contributions

We address the problem of supporting bi-temporal window queries. The technical challenge is the interplay between system and application time. System-time windows are essentially FIFO queues (tuples expire in arrival order) and can be maintained efficiently. However, while system time is append-only, application time can be arbitrary.

Indexing is a common solution for improving query performance but the state-of-the-art does not support bi-temporal indexing well. Bi-temporal window processing differs significantly from one-dimensional windows, simple bi-temporal queries and spatial queries. For example, bi-temporal tuples that are currently valid have unbounded system end times, but open intervals generally cannot be handled well by bounding boxes and space-minimizing partitioning strategies used in spatial indexing. Furthermore, there is little work on specialized bi-temporal indexing, and none that explicitly supports window queries (details in Section 3).

We make two contributions in this paper: 1) we introduce the concept of bi-temporal windows, and 2) we propose a two-dimensional index, BiSW, which supports a wide variety of bi-temporal window queries for both streaming and materialized data. Using a bi-temporal database benchmark based on TPC-H [13], we experimentally demonstrate the advantages of BiSW compared to the state-of-the-art in terms of query performance, maintenance overhead and space footprint. While bi-temporal indexing and sliding window processing have been separately addressed in prior work, this is the first paper that combines these important concepts.

The remainder of this paper is structured as follows. Section 2 introduces bi-temporal windows; Section 3 discusses previous work; Section 4 describes the proposed BiSW index and Section 5 explains how to use it with bi-temporal window queries; Section 6 presents experimental results; and Section 7 concludes the paper.

2. BI-TEMPORAL WINDOWS

2.1 The Bi-temporal Data Model

We model a bi-temporal relation as a non-temporal relation with four additional temporal attributes, $StartApp$, $EndApp$, $StartSys$ and $EndSys$, which define two time intervals. The $[StartApp, EndApp)$ application time interval represents an application-defined validity period of a tuple in the real world. The $[StartSys, EndSys)$ system time interval indicates when the tuple was visible in the database (until modified or deleted).

System time attributes are managed by the database system, while application time is managed by the user. A new tuple is given a $StartSys$ equal to the system time when it was inserted and $EndSys$ equal to infinity. If a tuple is deleted or any of its attributes (application time or other non-temporal attributes) are modified, the old version of it remains in the database, but its $EndSys$ is set to the system time when it was deleted or modified.

Table 1 shows an excerpt from a bi-temporal customer table corresponding to eight versions of a tuple with the same non-temporal key[1]; for brevity, we omit the non-temporal key itself and only include a unique tuple ID (TID), an account balance, and the application and system time intervals. The first tuple indicates that at system time 100, the account balance from application time 10 onwards was known to be 50. This information was valid until system time 102, when the second and third tuples were inserted (and at that time, the system end time of the first tuple was updated from infinity to 102). The second and third tuples indicate that at

[1]Of course, in practice a bi-temporal table contains many versions of tuples with many different non-temporal keys.

Table 1: An excerpt from a bi-temporal table

TID	Balance	StartApp	EndApp	StartSys	EndSys
1	50	10	∞	100	102
2	50	10	11	102	∞
3	40	11	∞	102	105
4	30	11	13	105	∞
5	100	13	15	105	106
6	30	15	∞	105	106
7	35	15	∞	106	∞
8	90	13	15	106	∞

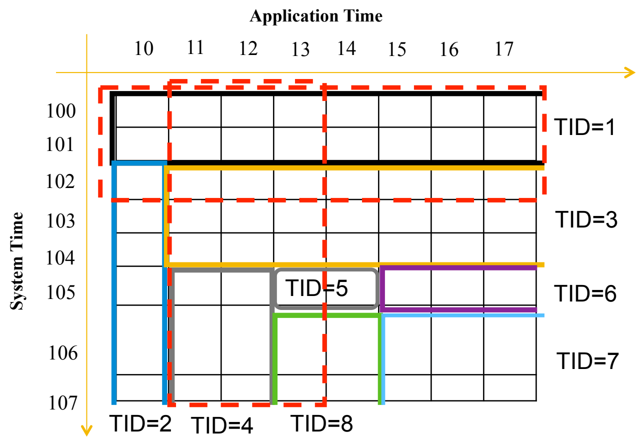


Figure 1: A rectangle representation of the data from Table 1 (solid lines) along with two example windows (dotted lines).

system time 102, the account balance was known to be 50 between time 10 and 11, and 40 from time 11 onwards. The second tuple has an open system end time, meaning that it is still valid. Tuples 4 through 6 were inserted at system time 105 with new information about the balance from time 11 onwards. Finally, tuples 7 and 8 were inserted at system time 106 with updated information about the balance from time 13 onwards; the balance before time 13 was recorded earlier (tuples 2 and 4) and has not changed since then.

Figure 1 shows a geometric representation of the system and application times of the tuples in Table 1, with application time on the x-axis and system time on the y-axis. Each tuple is represented by a rectangle, which may be closed or partially open. Ignore the dotted rectangles for now.

2.2 Bi-temporal Slicing

There are two ways to "slice" a bi-temporal table. One is along the system time dimension, which corresponds to snapshots of the table as recorded at different points in time, and to horizontal slices in Figure 1. Let t_{sys} be a system timestamp. The slice at t_{sys} corresponds to all tuples t where $t_{sys} \in [t.StartSys, t.EndSys)$. For instance, the slice at system time 104 includes tuples 2 and 3: if we draw a horizontal line at system time 104 in Figure 1, it intersects with TID=2 and TID=3. That is, if we roll-back the database to system time 104, the account balance was known to be 50 from time 10 to 11 and 40 from time 11 onwards. As new transactions arrive, we obtain new system time slices corresponding to new snapshots of the table.

The other possibility is to slice along the application time dimension, which corresponds to how our knowledge of the facts at different application times evolved, and to vertical slices in Figure 1. Let t_{app} be an application timestamp. The slice at t_{app} corresponds to all tuples t where $t_{app} \in [t.StartApp, t.EndApp)$. For example, the slice at application time 12 includes tuples 1, 3 and 4: if we draw a vertical line at application time 12 in Figure 1, it intersects with TID=1, TID=3 and TID=4. This tells us how our knowledge of the account balance at time 12 changed over (system) time: at system time 100, the balance was known to be 50; at system time 102 the balance was updated to 40; and at system time 105 the balance was updated to 30. As new transactions arrive, existing application time slices may grow (e.g., a new transaction may insert a tuple with an old application time), and new application time slices are created as needed.

Finally, we can ask a *point query* to find out what was known about a particular application time at a particular system time. For example, at system time 105, the account balance at application time 12 was known to be 30 (tuple 4). In Figure 1, this corresponds to drawing a horizontal line at system time 105 and a vertical line at application time 12, which intersect with TID=4.

2.3 Bi-temporal Windows

To cover a broad range of applications, we consider two definitions of bi-temporal windows, one based on interval overlap and one based on event containment.

2.3.1 Interval-Oriented Windows

Interval-oriented windows extend the slices we discussed earlier. Fixed windows directly correspond to collections of slices. Let t_{sys} and t'_{sys} be two system timestamps, with $t_{sys} < t'_{sys}$. A system time window $[t_{sys}, t'_{sys}]$ contains tuples whose system time intervals $[StartSys, EndSys)$ overlap the window. That is, the system time window identifies tuples that existed in any snapshot of the database between t_{sys} and t'_{sys}. For example, the system time window $[100, 102]$ includes tuples 1, 2 and 3, or the union of all tuples in system time snapshots at times 100, 101 and 102. This window corresponds to the horizontal dotted rectangle in Figure 1, which intersects with TID=1, TID=2 and TID=3.

Similarly, an application time window $[t_{app}, t'_{app}]$, for $t_{app} < t'_{app}$, contains tuples whose application time intervals $[StartApp, EndApp)$ overlap the window. This gives all the versions of tuples with applications times between t_{app} and t'_{app} that ever existed in the database. For example, the application time window $[11, 13]$ includes tuples 1, 3, 4, 5 and 8, or the union of all tuples in application time snapshots at times 11, 12 and 13. This window corresponds to the vertical dotted rectangle in Figure 1, which intersects with the above tuple IDs.

A *fixed bi-temporal window* includes both a system and an application window. For example, we can ask for all tuples within a system time window $[100, 102]$ and an application time window $[11, 13]$, which gives the intersection of the two result sets listed above, i.e., tuples 1 and 3. In Figure 1, this bi-temporal window corresponds to the intersection of the two dotted rectangles, which intersects with TID=1 and TID=3. Note that the order in which we apply the two windows does not matter: we can find the tuples within the system time window and then apply the application time window, or vice-versa.

The bi-temporal sliding windows we consider in this paper are defined by a *length* and a *slide interval* that indicates how often the window moves. We consider three types of sliding window queries.

The first involves a fixed system time window and a sliding application time window. Here, we identify a system time range of interest and we slide over application time. For example, suppose the fixed system time window is $[100, 102]$ and the sliding application window has a length of 3 and a slide interval of 1, starting at application time 13. The first result of the query contains all tuples within the system window $[100, 102]$ and the application window $[10, 12]$, which gives tuples 1, 2 and 3. Next, we move the application window to $[11, 13]$ and output tuples 1 and 3, followed by moving the application window to $[12, 14]$, and so on while keeping the system window fixed. In Figure 1, this corresponds to keeping the horizontal dotted rectangle fixed and moving the vertical dotted rectangle to the right, one application time unit at a time. As the vertical rectangle moves, the intersection of the two rectangles also moves to indicate the new contents of the window.

In the second case, the application window is fixed and the system window slides (in the special case of continuous queries, the system window ends at the current time and slides one *slide interval* at a time as new data arrive; otherwise, the system time window may start anywhere in the past and slides over the data currently in the database). That is, we identify an application time range of interest and report how the contents of the database have changed over (system) time. For example, suppose the fixed application window is $[11, 13]$ and the sliding system window has a length of 3 and a slide interval of 1, starting at time 103. The first result of this query contains all tuples within the application window $[11, 13]$ and system window $[100, 102]$, i.e., tuples 1 and 3. Next, we slide the system window to $[101, 103]$, giving tuples 1 and 3 again, followed by sliding the system window to $[102, 104]$, which gives tuple 3, and so on. In Figure 1, this corresponds to keeping the vertical dotted rectangle fixed and moving the horizontal dotted rectangle down, one system time unit at a time.

In the third case, both windows slide. Suppose both windows have lengths of 3 and slide intervals of 1, starting at system time 103 and application time 13. The first result of this query contains tuples within system window $[100, 102]$ and application window $[10, 12]$, which gives tuples 1, 2 and 3. Next, we move both windows forward by one, giving a system window $[101, 103]$ and an application window $[11, 13]$, which gives tuples 1 and 3. Next, both windows slide by one again, giving a system window of $[102, 104]$ and an application window $[12, 14]$, which only includes tuple 3, and so on. In Figure 1, this corresponds to moving both dotted rectangles one unit at a time.

2.3.2 Event-Oriented Windows

We also consider another definition of bi-temporal windows which we refer to as *event-oriented*. Let t and t' be two timestamps, with $t < t'$. An event-oriented system time window $[t, t']$ contains tuples with $StartSys$ or $EndSys$ in $[t, t']$; similarly, an event-oriented application time window $[t, t']$ contains tuples with $StartApp$ or $EndApp$ in $[t, t']$. That is, an event-orient system time window only contains those tuples which were inserted, modified or deleted between system time t and t', and an event-oriented application time window only contains tuples whose application time intervals *started* or *ended* between application time t and t'. These types of windows are useful in database auditing and tampering applications, to extract data that were modified during some period of time. Clearly, an event-oriented window includes a subset of the tuples from its interval-oriented counterpart.

Returning to Figure 1, an event-oriented system time window $[105, 106]$ contains tuples 3 through 8, i.e., those with system time ranges that start or end at time 105 or 106. Tuples whose rectangles intersect the window rectangle but are not bounded at least once in the window (e.g., TID=2) are not included.

2.4 Problem Statement

We now state the problem we want to solve. We are given a bi-temporal table, which may be static or continuously receiving new data, and we want to support queries with 1) fixed bi-temporal windows and 2) the three cases of sliding bi-temporal windows defined above, both interval and event-oriented, and both over streaming and historical data. We are interested in retrieving all tuples from a bi-temporal window range rather than searching for particular keys[2]. The specific problems we address are 1) inserting new data, 2) indexing the bi-temporal table to efficiently identify tuples belonging to a particular system and application time window, and 3) incrementally recomputing the set of relevant tuples when one or both windows slide. The second problem refers to fixed window queries and to computing initial windows for sliding window queries; we refer to these tasks as *range queries*. The third problem refers to sliding window maintenance and we refer to these types of tasks as *change queries*. The index should have a low maintenance overhead when new data arrive, a low space footprint and the means to easily partition or expire unneeded data.

3. RELATED WORK

Relational systems such as IBM DB2 [23], Oracle [21] and Teradata [1] have recently started supporting some bi-temporal operations. However, bi-temporal support is limited and bi-temporal operators are not optimized.

While the proposed BiSW index is the first bi-temporal index designed for window queries, there are alternatives [25]. Any spatio-temporal index structure can be used—as we showed in Figure 1, a bi-temporal tuple can be represented as a rectangle. Examples include variants of B-trees (see, e.g., [2, 7, 8, 20]) and R-trees (see, e.g., [5, 6, 18, 24, 29]). However, spatio-temporal indices do not cope well with the open intervals occurring in bi-temporal data, do not exploit the append-only nature of system time or suffer from the high cost of index maintenance.

Temporal indices have been proposed to deal with the above shortcomings. Two examples are the *Timeline Index* [14] and the *Bi-temporal Timeline Index* [12]. The timeline index is a main-memory log/change based index, but it only supports system time and exploits its append-only properties. The idea is to divide system time into buckets and log the transactions that have occurred in each bucket. The bi-temporal timeline index maintains a timeline index for system time in the same way as in [14]. Additionally, for selected system time buckets, timeline indices are created for application time, which index the application times of tuples that were valid at the given system time. If a query needs to look up an application time slice as of some system time that does not have an associated application timeline index, a new application timeline index is lazily built. This means that using the bi-temporal timeline index to support bi-temporal sliding windows requires a full application index construction for every system time slide.

The proposed index, BiSW, shares the idea of change indexing with [12, 14], but it is a two-dimensional change index. Each entry in the index records changes in both time dimensions, providing (almost) symmetric access for querying, but exploiting the semantic differences between system and application time for updates. As a result, it is more effective and less complex than fully symmetric spatial indexes for range queries and optimized for sliding window queries.

In terms of window queries, there is a great deal of previous work in data stream management; see, e.g., [11] for an overview. However, data stream systems do not fully support bi-temporal data. While bi-temporal stream models [3] and stream revisions [22] have been proposed, optimizing bi-temoral window queries was not discussed. Some data stream systems employ system or arrival time, while others rely on external application timestamps. In either case, stream tuples are assumed to arrive mostly in timestamp order, and out-of-orderness is handled by techniques such as buffering [31], overflow chaining [16] and punctuations [28, 30]. Having one time dimension with append-only arrivals greatly simplifies window maintenance and enables incremental processing; see, e.g., [15, 19, 26]. Again, it suffices to partition the single time dimension into buckets; new data are inserted into the new bucket, and, to maintain a sliding window, we simply drop old buckets when new ones are created. However, bi-temporal windows require new solutions.

Finally, we point out related work on moving queries over spatial data (see, e.g., [9, 27]), in which the region referenced in the query changes over time, and the objective is to maintain the query result. Moving queries are similar to sliding window queries, but the challenges and solutions in the context of bi-temporal window queries are different.

4. THE BISW INDEX

We now introduce the BiSW index, starting with the logical design, which is a two-dimensional change log. Next, we explain the physical design, which implements the two-dimensional change grid as an application-time-partitioned sparse matrix. We then present checkpointing optimizations that trade off storage space for lookup times, and discuss the space and maintenance complexity.

4.1 Logical Design

When modelling bi-temporal data, it is useful to understand which classes of records exist and how they evolve. There are four classes of bi-temporal tuples, corresponding to four different shapes, discernible by their end values in each temporal dimension:

1. Rectangles with two open sides, corresponding to tuples with unbounded system and application end times; e.g., TID=7.
2. Horizontal half-open stripes, corresponding to tuples with unbounded application end times but bounded system end times, meaning that a newer version of this tuple exists; e.g., TID=1.
3. Vertical half-open stripes, corresponding to tuples with unbounded system end times, but bounded application end times; e.g., TID=2.
4. Closed rectangles, corresponding to tuples with bounded system and application end times; e.g., TID=5.

Since application time updates are accompanied by a system time update and system time values are immutable (except for being closed when a new version of the tuple is created), the only updates that may happen to an existing tuple are (1) to (2) or (3) to (4). All other transactions result in a new tuple that again falls into one of the four classes.

Many of these shapes are open, so classical approaches such as R-trees that rely on closed bounding boxes and balanced space partitioning are not effective. Given that window processing is based on changes in the data, we take a different approach in BiSW by indexing the boundary points that define a shape.

The logical design of BiSW is that of a two-dimensional grid of cells, in which a cell with coordinates (x, y) holds events that occurred at application time x and system time y. To disambiguate the meaning of these events, we also store their "role". For this

[2]Using the "key/valid/transaction" terminology from [25], our queries are "*/range/range".

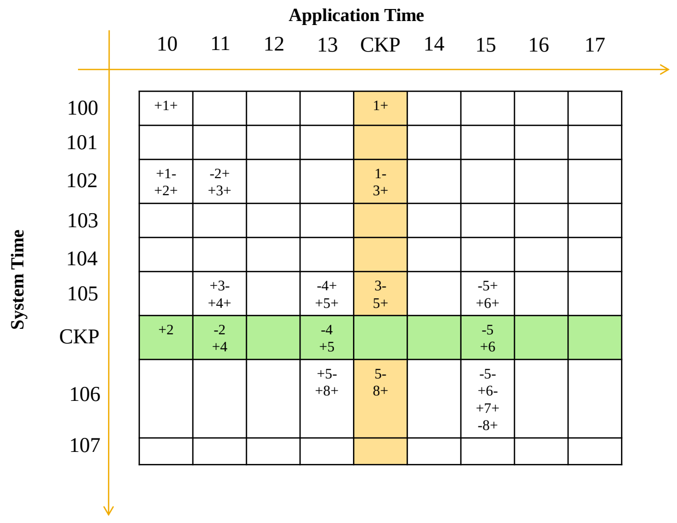


Figure 2: Representing the boundary points of bi-temporal tuples. The shaded strips correspond to checkpoints (see Section 4.3).

purpose we use the following notation: $opApp\ TID\ opSys$, where $opApp$ and $opSys$ can be $+$ or $-$, representing the beginning or the end of this tuple in application time and system time, respectively.

For example, consider TID=5 in Figure 1. Its starting point is (13,105), so this particular cell of the index stores $+5+$, denoting the beginning of application and system time intervals for tuple 5. In application time, this tuple spans to 15, so we store $-5+$ in cell (15,105). Similarly, since the system time of this tuple ends at 106, we store $+5-$ in cell (13,106) and $-5-$ in cell (15,106).

Figure 2 illustrates the logical view of BiSW given the data from Table 1. Ignore the shaded horizontal and vertical strips labeled CKP for now. Note that tuples from class 1 such as TID=7 (open system and application time range) only have one entry for them, tuples from classes 2 and 3 such as TID=1 or TID=2 (one open interval, one closed interval) have two entries, and tuples from class 4 such as TID=5 (closed in both times) have four entries[3].

4.2 Physical Design

While a spatial index could store our boundary points, exploiting order, storage locality, compression and efficient updates led us to a sparse matrix design that is partitioned by application time and relies on arrays of actually-present versions in system time. Figure 3 describes how the running example from Table 1 is (partially) translated and represented. For each application time, we store a sorted array of system times (labeled Version) along with the events that happened in the corresponding logical cell (Events). This allows efficient scans and selections of a specific version (by direct lookup or binary search over the version list). Note that cells that do not contain any events, such as application time 12 or system time 101, are not included in the physical implementation.

Depending on the distribution and granularity of application times, the application-time partitions can be maintained in different ways. First, a partition may not cover a single application time, but a range, thus reducing the number of partitions. If the application times are regularly distributed, a simple array pointing from application timestamps to system time arrays is suitable, as in Figure 3. In other cases (such as scientific applications), application times

[3]For class-4 tuples, the standard geometric interpretation calls for just the diagonal points, $+X+$ and $-X-$. However, as we will see in Section 5, we need three or all four points to answer bi-temporal window queries efficiently.

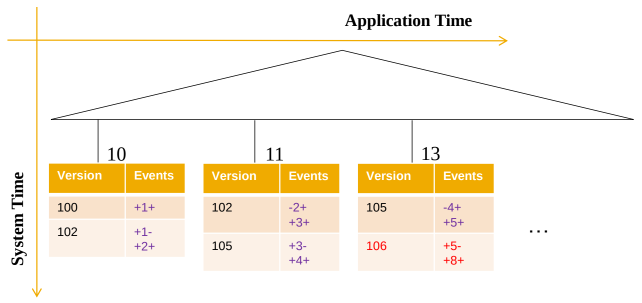


Figure 3: Physical design of BiSW

may have a higher resolution and a skewed distribution, and the application-time partitions may be stored in a balanced tree.

The append-only property of system time ensures that the system time arrays are only appended to, thus ensuring efficient updates. When new data arrive, they are partitioned on-the-fly by their application time, and appended to the end of the corresponding system time arrays. For the updates happening at system time sys_t, only those horizontal cells which share the same system time coordinate sys_t will be touched. For example, returning to Table 1, tuples with TID=7 and TID=8 are inserted into the database at system time 106. Also, note that tuple 7 is a new version of tuple 6 and tuple 8 is a new version of tuple 5, so we need to set the system end time of tuples 5 and 6 to 106. This triggers the following index operations at time 106: 1) append a cell with Version=106 under application time 13, and store $+5-$ and $+8+$ there, as illustrated in red in Figure 3; and 2) append a cell with Version=106 under application time 15, and store $-5-$, $+6-$, $+7+$ and $-8+$ there.

One may ask how to tell if newly inserted tuples are new versions of existing tuples, and therefore how to tell if any existing tuples must have their system end times updated. For this, we can maintain a separate index structure, such as a hash table, that maps non-temporal keys to TIDs. Note that we need this information not only for BiSW, but also for any other choice of index on the bi-temporal attributes.

4.3 Checkpointing

Indexing the boundary points avoids the overhead of indexing open shapes, enabled append-only updateds, and, as we shall see in Section 5, it provides an efficient way to maintain sliding windows by identifying changes between adjacent system and application time ticks. However, range queries such as those for finding the initial windows for sliding window queries, incur a higher overhead, since we need to check all the preceding cells in both time dimensions to find previously-opened intervals that overlap with the window. To speed up range queries, we introduce *checkpoints*, which summarize the events that have occurred so far.

Figure 2 illustrates a system time checkpoint after time 105 and an application time checkpoint after time 13, labeled CKP. The system time checkpoint corresponds to the green horizontal strip. It encodes the application start times ($+X$), and, if applicable, application end times ($-X$), of tuples that are valid at time 105, partitioned by application time. There are four such tuples: 2, 4, 5 and 6. All but 6 have a bounded application end time and therefore all but 6 have both a $+X$ and a $-X$ in the checkpoint. The application time checkpoint corresponds to the yellow vertical strip. It encodes the system start times ($X+$), and, if applicable, system end times ($X-$), of tuples whose application time range includes 13, partitioned by system time. There are four such tuples: 1, 3, 5 and 8, of which all but 8 have a bounded system time and therefore both an $X-$ and an $X+$.

Checkpoints trade off space for query performance; e.g., to find valid tuples at time 106, we refer to the illustrated checkpoint and the cells corresponding to system time 106 instead of scanning all the cells up to system time 106. To create a new checkpoint, we go back to the most recent available checkpoint and only examine the new events that have been recorded since then. One way to implement checkpoints is using bitmaps.

Checkpoints also give us the ability to partition and prune the index, since a checkpoint represents the full set of active tuples in a more compact way than the index itself. This way, even with continuous queries and tuples with arbitrary lifetime intervals, the amount of history that needs to be stored can be bounded.

4.4 Space Complexity

While BiSW logically employs a grid over the system and application times, its storage requirements are proportional to the number of bi-temporal tuples; recall that at most four events are required per tuple. This is expected to be much smaller than the cross product of the two time domains since it is unlikely that every system time will include transactions that update every possible application time. As illustrated in Figure 3, each system time array needs linear space in the number of versions it stores, and altogether the arrays store a number of events that is linear in the number of tuples. Partitioning by application time creates minimal overhead since it only requires entries for application times that actually exist. Depending on whether this partitioning is dense or sparse, the cost of the directory over the partitions is linear or logarithmic in the actual domain of application time. Finally, checkpoints need as many entries as there are active intervals. However, in practice, most intervals will not span the entire time range. Thus, the worst-case complexity is $O(kN)$ where k is the number of checkpoints and N is the number of tuples.

Compared to the Bi-temporal Timeline index and other approaches that fundamentally rely on single-dimensional indices, BiSW is more space-efficient (it does not maintain separate application time indices at different system time points and therefore does not store the same tuple multiple times) and incurs no overhead when looking up arbitrary combinations of system and application times. Compared to spatial indexes, BiSW avoids the impact of half-open shapes and therefore achieves better update and querying behaviour.

5. USING THE BISW INDEX

We now describe how to use the BiSW index to answer bi-temporal window queries. We use the following simple query, written in a format inspired by streaming SQL query languages, with “table” referring to Table 1:

```
select *
from table
    [SYS, START 101, RANGE 4, SLIDE 1]
    [APP, START 11,  RANGE 3, SLIDE 2]
```

The initial instance of this query runs over the system time window $[101, 104]$ and application time window $[11, 13]$. In the next instance, the system time window slides by one to $[102, 105]$ and the application window slides by two to $[13, 15]$, and so on. We can omit the START times if the windows are to end at the currently-largest system or application time and slide as new data arrive, as is the case in continuous queries.

We describe 1) how to use the index to answer the initial range query and 2) how to answer change queries that identify changes from one instance to the next, which may be used to incrementally maintain the window contents. Throughout this section, we will refer to Figure 4, which takes Figure 2 and adds three rectangles, corresponding to the initial instance of the bi-temporal window of the above query and two subsequent instances. We will explain the shaded regions shortly.

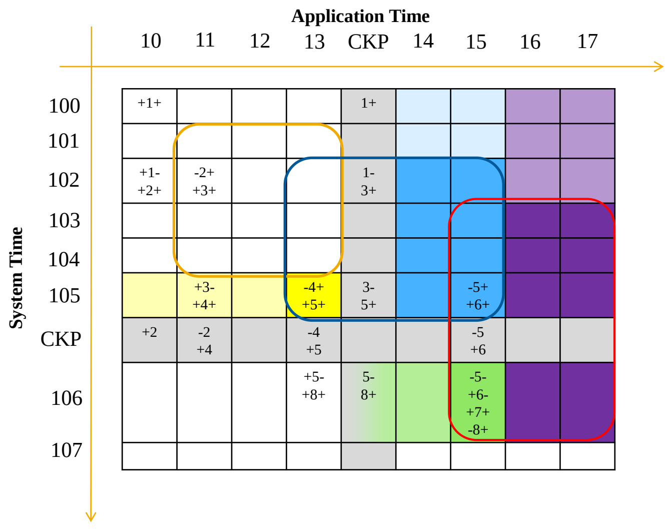


Figure 4: Windows arising from the example query.

5.1 Event-Oriented Windows

For event-oriented windows, computing the initial instance is simple: we report exactly those events which are stored in the cells spanning system time 101 through 104 and application time 11 through 13. These cells are outlined in yellow in Figure 4, and the initial events are $-2+$ and $+3+$. In terms of the physical design (recall Figure 3), we need to access the application time partitions 11 and 13, and for each, we access the system time arrays in positions 101 through 104. If we have an index on the application time partitions, we can locate the desired application start time in time logarithmic in the number of application time partitions. Then, we can use binary search to locate the desired system start time within a system time array.

Change queries are also simple. When the windows slide, we compute the negative delta, corresponding to *expired* events, i.e., those in cells that belonged to the previous instance of the windows but not the current instance, and the positive delta, which consists of new events in cells that did not belong to the previous instance. Again, this can be done by accessing the appropriate application time partitions in BiSW. In Figure 4, the next instance of the windows is outlined in blue (ignore the grey strips corresponding to the system and application time checkpoints, as we do not need checkpoints for event-oriented queries). The negative delta is $-2+$ and $+3+$ and the positive delta is $-4+$, $+5+$, $-5+$ and $+6+$.

Index lookup gives “raw” events, but their interpretation is straightforward. For example, the initial state of the query of $-2+$ and $+3+$ means that for TID=2, its application time interval ends within the window and its system time starts within the window; for TID=3, its system and application time intervals both start within the initial windows.

5.2 Interval-Oriented Windows

5.2.1 Range Queries

We now turn to interval-oriented windows, where it is not immediately obvious how to convert the events stored in the index into intervals. First, consider the initial range query and assume there are no checkpoints. To obtain the intersecting intervals, we find those which started within the window plus those which started earlier but have not yet ended. This corresponds to accessing cells within the initial window as well as cells above it and to the left. In Figure 4, to find events relevant to the first instance of the query (i.e., system time window $[101, 104]$ and application time window $[11, 13]$), we need to examine the top-left region of the cells all the way down to application time 13 and system time 104. These events are: $+1+$, $+1-$, $+2+$, $-2+$ and $+3+$.

Next, we explain how to construct intervals from events. The idea is to take the $+X+$ events (i.e., tuples 1, 2 and 3 in our example), but whenever we see a matching $-X+$ or $+X-$ event, check if the application or system end time, respectively, is still within the window. If not, we can immediately remove the corresponding tuple from consideration. Since event $+1-$ is in the cell $(102, 10)$, we know that the system end time of tuple 1 is 102, which is within the initial range query. On the other hand, $-2+$ is in the cell $(11, 102)$, meaning that the application end time of tuple 2 is 11, which is *not* within the initial range query[4]. Thus, we only report tuples 1 and 3 in the initial query result.

To perform the reconstruction efficiently, it is crucial to scan the index in order: we proceed in application time order and scan each associated system time array in order. For each plus-plus event, we add the corresponding tuple to a list of candidates. When we encounter a corresponding minus-event, we check the system or application end time of the event, as described above, and decide whether to keep the tuple in the candidate list of remove it. Note that the end times can be obtained from the index alone by examining the cell co-ordinates of the corresponding event, and we do not have to access the bi-temporal table.

There is one more piece of information that our range query lookups return, which is useful for subsequent change queries. As we are scanning the events, if we decide to keep a tuple in the candidate list after seeing a minus-event, we also record the tuple's system or application end time in the candidate list. In the above example, tuple 1 will be reported in the answer and it will be marked as ending at system time 102. If the range query is just a fixed window query, it can ignore this additional information.

5.2.2 *Change Queries*

When a bi-temporal sliding window query is issued, we first perform a range query and then we repeatedly perform change queries whenever the window or windows slide. Throughout the execution, we need to maintain the result set along with the extra information about end times, as described above. We now discuss the first window-slide of our example query, i.e., the system time window slides to $[102, 105]$ and the application time window slides to $[13, 15]$, giving the rectangle outlined in blue. We know that the initial range query returned tuples 1 and 3, and we know that tuple 1's system time interval ends at 102. Our goal now is to calculate the positive and negative delta.

The first step is to find expired events, i.e., the negative delta. For this, we examine the previous result set and see if any tuple's end time is now outside the new window(s). Tuple 1's system end time of 102 is outside the new system time window $[102, 105]$, so tuple 1 is added to the negative delta and removed from the result.

To improve the efficiency of computing the negative delta, we should not scan the entire result set. One simple optimization is to partition the result set: one partition stores tuples with no minus-events, the next partition stores tuples that will expire one slide from now, the next partition stores tuples that will expire two slides from now, and so on. For example, after computing the initial range query, it is easy to compute that tuple 1's system end time of 102 means that tuple 1 will expire one slide later.

The second step is to obtain new events. For this, we access the cells within the new windows, and some additional cells. Those additional cells are shaded light yellow and light blue in Figure 4: the horizontal strip for system time 105, from the beginning of applicaton time till 12, and the vertical strip for application times 14 and 15, from the beginning of system time till 101. To understand why scanning these additional regions is necessary, observe that there may be a tuple with a system start time of 105 but an old application time, which did not overlap with the initial window but now overlaps with the current window. We would miss these tuples if we only examined the cells within the new windows. From these regions, we obtain the following new events: $+3-$, $+4+$, $-4+$, $+5+$, $-5+$ and $+6+$.

We now apply the same logic as in range queries and check the end times of each minus-event to decide whether to keep it in the candidate list or remove it. The system end time of tuple 3 is 105, so this tuple remains in the candidate set. Similarly, the application end time of tuple 5 is 15, so it also remains in the candidate set. On the other hand, the application end time of tuple 4 is 13, which is not within the new application time window, so this tuple is removed from the candidate set. This gives a positive delta of tuples 5 and 6 (tuple 3 was already in the previous result set).

We now have all the information we need to update the result of the query after the first slide. The old result, containing tuples 1 and 3, the negative delta of tuple 1, and the positive delta of tuples 5 and 6 give a new result with tuple 3 (and its system end time of 105), tuple 5 (and its application end time of 15), and tuple 6. Again, it is easy to compute that tuple 3 will expire three slides from now and tuple 5 will expire one slide from now.

After the third slide, the system time window is at $[103, 106]$ and the application time window is at $[15, 17]$. We put tuple 5 in the negative delta because its application end time is outside the new application time window. Next, we scan the cells contained in the new windows (outlined in red) as well as the regions shaded in light purple and green[5]. The new events are: $+5-$, $+8+$, $-5-$, $+6-$, $+7+$ and $-8+$. We also notice that the application end time of tuple 8, which is 15, does not overlap with the new application time window and we remove tuple 8 from consideration. Tuple 6's system end time of 106 is still in the window, so it remains in the result. Finally, tuple 7 is added to the positive delta. Thus, the updated result consists of tuples 3, 6 and 7.

Note that the above process applies to both continuous and historical queries, with the only difference being that for historical queries, all the data in Figure 4 already exist, whereas for continuous queries, the grid in Figure 4 is being created as new data arrive, one horizontal strip at a time. In both cases, we scan the same regions when the windows slide: the horizontal strip corresponding to new system times and the vertical strip corresponding to new application times.

5.2.3 *Exploiting Checkpoints*

Figure 4 shows a system time checkpoint at time 105 and an application time checkpoint at time 13. Checkpoints are very useful for range queries. For example, suppose the initial windows are as outlined in red, i.e., system time $[103, 106]$ and application time $[15, 17]$. Rather than scanning all the cells above and to

[4]Recall that tuple intervals are closed on the left and open on the right, therefore tuple 2's application time interval of $[10, 11)$ is not within the application time window of $[11, 13]$.

[5]We do not have to scan all the cells with system time 106 because of the checkpoint at application time 13. That is why the green interval does not extend all the way to the left.

Table 2: Data Set Properties

Data Set S	SF_0	SF_H	#tuples	#app.versions	#sys.versions
1	1	10	3 Mio	3 Mio	3 Mio
5	1	50	17 Mio	15 Mio	15 Mio
10	1	100	57 Mio	55 Mio	55 Mio

the left of this initial region, it suffices to go back till the nearest checkpoint and only scan the remaining cells. Returning to the physical implementation of BiSW, this can greatly reduce the number of application time partitions and the corresponding system time arrays that need to be accessed. Again, by examining the end times of minus-events, the checkpoints tell us that tuples 3, 6 and 8 are candidates for the result, and all we have to do next is access the cell $(15, 106)$ to find that 7 is also in the result set but 8 is not. Checkpoints can also be useful for change queries, as we saw in the earlier example.

6. EXPERIMENTS

6.1 Setup, Data Sets and Contenders

We performed the experiments on a server with 256GB RAM and 2 AMD Opteron 6276 CPUs with 16 cores running at 2.3 GHz. We used a Linux operating system with a 3.8.5 kernel. We repeated each measurement ten times and report the average. Any experiment taking longer than one hour was stopped.

For our datasets, we used the schema and data generator from the TPC-BiH [13] benchmark. The generator uses the output of the non-temporal TPC-H generator and adds a temporal dimension to it by running additional update transactions (such as new order, cancel order). Thus, while the update generator is running, the system time advances linearly. We modified the TPC-BiH generator slightly and chose the application time interval for each tuple from a uniform distribution. This modified TPC-BiH generator yields a dataset for which the events are distributed uniformly in the application and system time domains. The size of the dataset is influenced by two scaling factors: SF_0 is the scaling factor of the non-temporal TPC-H generator used for initialization and SF_H determines the size of the history in terms of Millions of update transactions. Table 2 summarizes the three data sets we generated in terms of the number of tuples, the number of distinct application times and the number of distinct system times. The queries were run on the orders table which has predominantly closed application times (most orders were completed at some point) and open system times (orders stay in the system).

We implemented BiSW in C++ as a stand-alone, single-threaded prototype and compared it with the following alternatives.

- **BiTL [12]** keeps a change log in the system time dimension and creates application time change logs for selected system time snapshots. Queries that only slide on application time use the appropriate application time change logs. Queries that slide only the system time or both times use the system time change log and multiple application time change logs.

- **B+-Tree.** We used two independent B+ Trees, one for each time dimension, from panthema.net/2007/stx-btree. Similar to BiTL and BiSW, the B+ Trees index the start and end points of intervals. The key in the B+ Tree is a time point and the value is a pointer to a set that stores all the TIDs with start/end points at that time point. We use the sign of a TID to indicate openness of the interval, i.e., a positive TID indicates an open interval. We used the default fan-out of 128, thus each node had at most 2KB size. Bi-temporal queries are answered by independently computing the tuple IDs satisfying each window range (or change) and then intersecting them. If only one time dimension is sliding and the other is fixed, only one tree needs to be accessed.

- **RR*-Tree [4]**. Here, a 2-dimensional R-tree, with 8KB node size, indexes the system and application time intervals. We set all unbounded end times to the largest possible value. We implemented bi-temporal window queries (as rectangle queries) using the libraries from www.mathematik.uni-marburg.de/seeger/code/rrstar.

- **Table Scan.** In addition to index structures, we also compare the performance of scanning the four temporal attributes (start and end times in each time dimension), vertically partitioned into a column store. To ensure a fair comparison with the append-only behaviour of BiSW, we keep the data in system time order.

6.2 Lessons Learned

Before presenting detailed results, we summarize our main experimental findings below.

- BiSW wins in query performance, often by a significant margin. Some competitors come close for specific types of queries or parameter settings, but none can compete overall. On update performance it comes second after BiTL, but the margin is small. Table (actually Column) Scan as a baseline is often quite competitive, since it uses scans on compact, contiguous data, which is a good fit to modern hardware. Furthermore, the data points are sorted in system time, so early stops are possible.

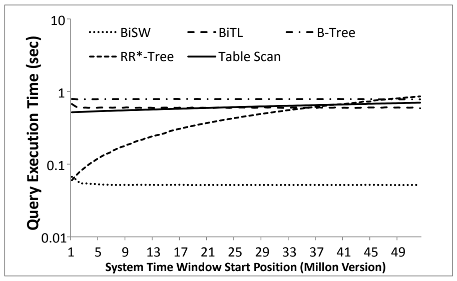
- BiSW is the only method that is viable for continuos queries, consistently outperforming all competitors.

- Temporal skew has a major impact on query performance, maintenance cost and storage requirements, but different techniques are affected in different ways. Open intervals cause RR* to suffer from bad selectivity/partitioning and therefore poor query performance. BiSW has good query performance even with many open intervals, but checkpoints use more space because open intervals lead to tuples that are valid throughout history and are stored in many checkpoints.

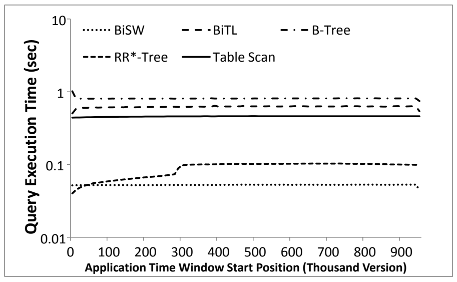
- Since it stores rectangles and not events, the RR*-Tree does not give better performance for the simpler event-oriented windows. Storing events as points is possible, but this would in turn limit the performance for interval-oriented windows.

- For interval-oriented windows, the query performance of BiSW depends on the position of the window in the system and application time space because we must access some cells from the beginning of time till the current window. Fortunately, even with a small number of checkpoints, we can reduce query costs significantly. However, checkpoints increase the space and maintenance overhead of BiSW, especially if there are many tuples with long or open intervals.

- BiSW handles sliding on system time slightly better than sliding on application time. This is because sliding on system time while keeping the application time window fixed does not require accessing new application-time partitions and their corresponding system time arrays of events. However, BiTL handles sliding on application time better because sliding on system time requires BiTL to create a complete application time index whenever system time slides.

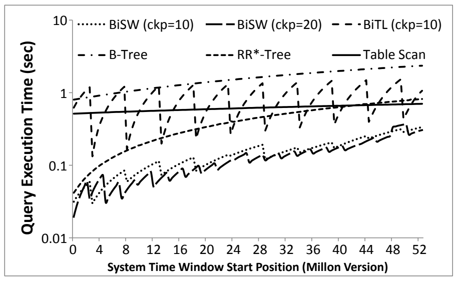


(a) System Time

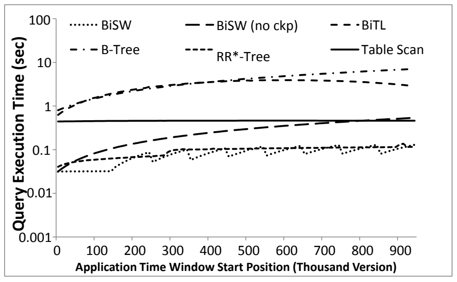


(b) Application Time

Figure 5: Experiment 1: Query performance for fixed event-oriented windows as a function of the starting position of the window.



(a) System Time



(b) Application Time

Figure 6: Experiment 1: Query performance for fixed interval-oriented windows as a function of the starting position of the window.

6.3 Experiment 1: Fixed Window Queries

The first set of experiments describes range queries over fixed bi-temporal windows. First, we vary the position of the fixed window in the bi-temporal value space. We measure the query time over a window that spans 5 percent of the value space in both dimensions on the *S=10* data set (refer to Table 2 for details). Figures 5 (event-oriented) and 6 (interval-oriented) show the effects of moving the starting position of the window in system time (a) and application time (b) while keeping the other in the middle of the value space. The logarithmic y-axes show the query running time.

For event-oriented windows (Figure 5), BiSW wins and is not affected by the starting position of the window (we can go to any starting position by accessing the appropriate application time partition). RR* comes the closest, but it slows down as the starting position moves away from the beginning, because it has to index an increasing amount of open intervals which it cannot handle well.

For interval-oriented windows (Figure 6), all but RR* and Table Scan consider prefixes of the window to find intervals that opened before the window started, and therefore query times get worse as the starting position moves. BiSW and BiTL (in the system time dimension) benefit from checkpoints; "(ckp=x)" denotes the number of uniformly-distributed checkpoints. For BiTL, the number of checkpoints corresponds to the number of application time indices. In general, BiSW and RR* are the only indexing methods that can substantially outperform a table scan, as they can exploit the bi-temporal/spatial properties of the data. B-Trees can only handle a single dimension well, whereas BiTL suffers from the cost of constructing multiple application time indices.

Since interval-based windows are more challenging and have more complex performance behaviour, we will focus on them in the rest of this section.

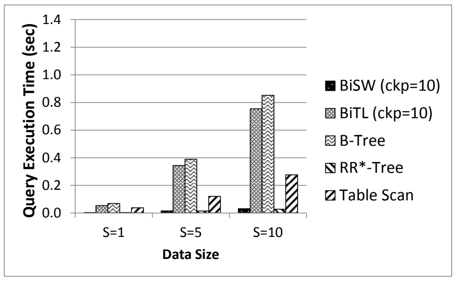
We now illustrate scalability in Figure 7, given a fixed window (a) close to the beginning of the value space (starting around 5% of the space in both dimensions) and another (b) in the middle of the time value space. The x-axes shows three different data set sizes and the y-axes measure the query execution time. The first case is generally cheap (small prefixes, few open intervals), but does not permit the use of checkpoints. The second is more expensive overall, but checkpoints are helpful. In both cases, B-Tree and BiTL are worse than Table Scan; in the second case also RR*.

6.4 Experiment 2: Sliding on System Time

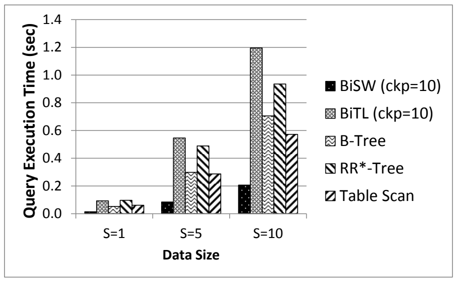
The second set of experiments focuses on the query cost with a sliding window on system time. The cost depends on the fixed window size in application time, the slide interval and the position of the window. Figure 8 shows the effect of each of these parameters, again on the large data set with an initial window as in the first experiment. When varying the fixed application-time window size (Figure 8a), we expect the slide cost to increase, as a larger area needs to be covered for each slide. BiSW shows exactly this behaviour, and outperforms BiTL by a factor between 3 to 8, since it can directly access the events needed for the delta. RR* and Table Scan are significantly slower and not affected by the size. For an increasing window slide (Figure 8b), we again see the expected increase in cost when the area gets larger and more intervals are retrieved. BiTL is affected the most since it needs to create an increasing number of application time indexes, making it very slow. In absolute numbers, BiSW is again the clear winner (up to a factor of 16 faster than RR*), but its lead diminishes if the slides get very big. The position of the sliding window (Figure 8c) has a similar effect as the position of a fixed window (Figure 6b), while BiSW is now even more effective since only the slide area in one prefix dimension needs to be checked. Finally, when varying the scale of the data sets, all methods scale equally well, retaining their relative performance ratios. Due to space constraints, we omit the figure.

6.5 Experiment 3: Sliding on Application Time

Figure 9 shows the results corresponding to Figure 8, but for a sliding application-time window. The results are similar to the previous experiment, but BiTL and RR* now perform better—the former because it does not have to construct multiple application-

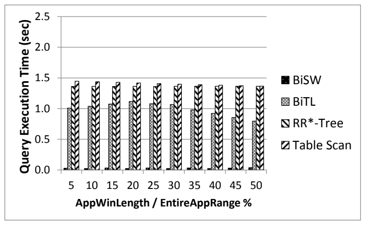


(a) Beginning of Time

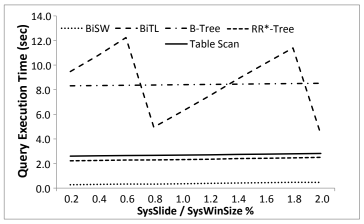


(b) Middle of Time

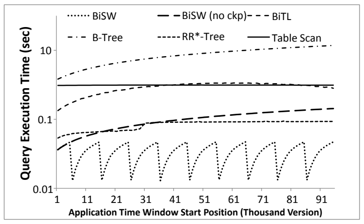
Figure 7: Experiment 1: Scalability with respect to data set size for fixed windows.



(a) Size of app. time window



(b) Slide interval



(c) Position of app. time window

Figure 8: Experiment 2: Effect of various parameters on query performance for a sliding system time window.

time intervals and the latter because our data set contains fewer open application time intervals than system time intervals.

6.6 Experiment 4: Sliding on Both Times

The fourth set of experiments covers windows that slide on system and application time. Figure 10 shows the effect of three parameters on the query execution time: window size, ratio of the system to application time slide interval and data set size. In terms of window sizes, BiSW wins again. An interesting insight can be gained by modifying the slide ratio (Figure 10 c): RR* and Table Scan are unaffected, as they perform the same operations regardless of which time dimension slides. BiSW gains about a factor of three in performance when the slide ratio changes from mostly on application time to sliding mostly on system time, since it can now always access the same application-time partitions. BiTL gets worse by almost a factor of six since more and more application time indices need to be created. We also evaluated the slide interval. Since the results indicate no new insights over the previous experiments, we omit the figure.

6.7 Experiment 5: Data Skew

Given the partitioning approach of BiSW, the granularity of time in both dimensions may have a measurable impact on performance, since it affects how much data will be accessed in any one partition. Similar effects may be present for the other index types. To investigate this effect, we modified our existing data set to reduce the number of versions (but not data items) in each dimension and thus either reduce the number of partitions (for application time) or the length of each partition (for system time).

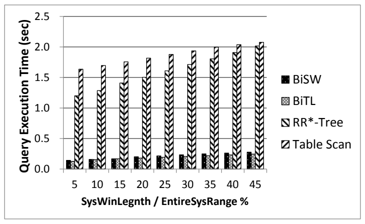
Figure 11 shows the results of reducing (a) application times and (b) system times down to 100, ten and one. When the number of application times decreases, both BiSW and BiTL see a cost increase, since the application time is no longer selective (BiSW) or provides room for indexes (BiTL). The other methods are not affected. When the number of system times is decreased, the cost increases for all approaches: BiSW and BiTL can no longer perform long scans over the system timeline sequences, thus losing efficiency. Table Scan loses performance since it can no longer be easily limited to a part of the system-time sorted data, while RR* has to contend with little selectivity in the system time domain.

6.8 Experiment 6: Space and Maintenance Overhead

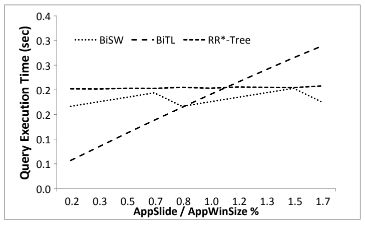
Next, we investigate the cost incurred by the index in terms of space consumption and maintenance time.

The results in Figure 12 (a) show that both BiSW and BiTL are very compact without checkpoints, requiring 9 and 6 % of the space used by the table, while a B-Trees requires 30 % and RR* 40 %. Adding checkpoints increases the memory consumption considerably, as each checkpoint needs to represent all active intervals. For application time, which consists mostly of closed-ended intervals, the overhead is around a factor of 3 for 10 checkpoints; but for system time, which consists mostly of open-ended intervals, this leads to an increase by a factor of 5 for 10 checkpoints. Providing checkpoints in both dimensions and doubling the number of checkpoints thus lead to the expected results, so that eventually BiSW requires more storage than the actual table. It should be noted, however, that the implementation of the checkpoints is not at all optimized, using STL sets at every cell. Adaptive compression (switching between different representations) and incremental checkpoints are likely to yield significant reductions in memory consumption. When scaling the data set size (Figure 12 (b)), all index types experience a similar effect. In order to assess the space overhead of the application time partitions, we modified the ratio of application times changes per system time changes. The results showed that the space consumption was affected to a very small degree.

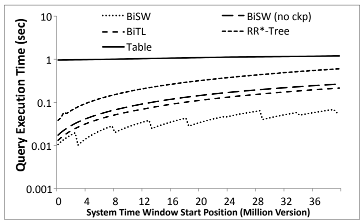
While the previous experiments considered query processing costs, we now examine the maintenance overhead of the different indices. For each, we insert the entire data set and measure time to completion. In Figure 12 (c), we show the effect of scaling the data set. BiSW (and also BiTL and B-Tree) perform quite well, and we will also see in the next experiment, the additional cost of checkpoints is fairly moderate. RR* is more than 1.5 orders of magnitude slower.



(a) Size of sys. time window

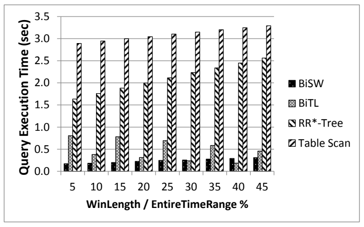


(b) Slide Interval

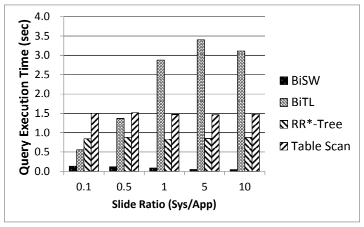


(c) Position of sys. time window

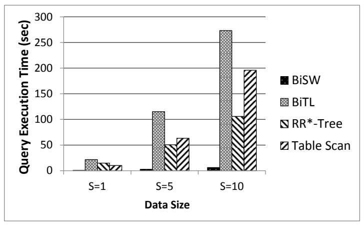
Figure 9: Experiment 3: Effect of various parameters on query performance for a sliding application time window.



(a) Size of window



(b) Ratio of slide intervals



(c) Data set size

Figure 10: Experiment 4: Effect of various parameters on query performance for a sliding both windows.

6.9 Experiment 7: Continuous Queries

Our experiments so far focused on static data, separately investigating query performance and index maintenance costs. However, sliding windows are commonly used in continuous queries where data ingest and query processing happen simultaneously. To efficiently support such workloads, index performance must not degrade significantly with the amount of data. In Figure 13, we show the throughput (in millions of tuples per second) that the different techniques can achieve on two common continuous query workloads: sliding windows on both time dimensions (a) and sliding windows on system time only (b). The data set used in this experiment contains 200 thousand tuples per system time value (referred to as a batch), and the window slides whenever the system time is incremented. The queries are started only once we have enough data for complete window instances, so up to batch 6 we only see data loading.

The first case (Figure 13 (a)) covers windows that slides in both system and application time with the incoming data, representing a consistent view on now-recent data that most streaming applications need. The results show that BiSW is the only index structure suitable for such a workload, as it can sustain a rate of around 1 million tuples/second and keep this performance constant over growing data. Given our currently naive implementation of generating checkpoints by "stopping the world", we see brief slowdowns, but this can be easily overcome by continuously collecting the information needed for checkpoints or delegating this task to separate threads. BiTL fluctuates significantly more due to the cost for slides between application checkpoints, reaching only an average throughput of around 270K tuples/second. The higher indexing speed observed in Figure 12 (c) is more than offset by the much higher query processing cost. Not using an index at all gives even better ingest rates (around 9 million tuples/second with occasional drops due to reallocations), but querying by scanning all the data is even more expensive and degrades quickly. On average, 150K tuples/second can be processed. Finally, the high cost of indexing and the degrading performance with additional data make the RR*-tree unsuitable for continuous queries, providing an average throughput of only around 12K tuples/second. The B-Tree has very high loading rates, but the query performance is orders of magnitude slower so the results are not shown.

The second case (Figure 13 (b)) covers a window that is fixed in application time but slides over system time, providing a view of the revisions [22] of a particular time period. The overall trends follow the observations in the previous experiments, with some subtle differences: Since only one direction of sliding is needed, the B-Tree becomes more competitive, but its throughput still does not exceed 65K tuples/second. BiTL also sees slight speedup to around 290K tuples/second since application times only need to be checked, but no temporary indexes need to be built. Similarly, Table Scan needs to check only application time attributes, and therefore fewer columns need to be accessed.

7. CONCLUSIONS

In this paper, we initiated a study of bi-temporal windows, which bring together the two important concepts of bi-temporal databases and window queries over temporal attributes. We formulated and solved a novel problem in this context: indexing to support queries with fixed and sliding bi-temporal windows, over historical and streaming data. We proposed and experimentally evaluated a new two-dimensional index, called BiSW, that outperforms the state-of-the-art in terms of lookup speed, maintenance overhead and space complexity. There are many open problems in the context of bi-temporal windows that we plan to study in future work, including query optimization, benchmarking, and integrating bi-temporal support in data stream systems and data stream warehouses.

8. REFERENCES

[1] M. Al-Kateb et al. Temporal Query Processing in Teradata. In *EDBT*, 573–578, 2013.

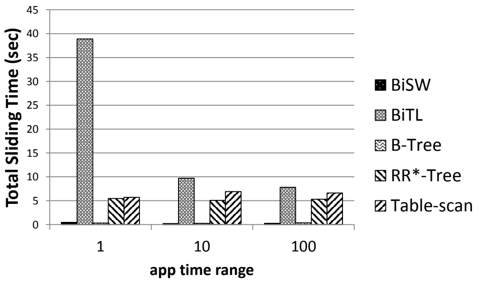
[2] C.-H. Ang and K.-P. Tan. The Interval B-Tree. *Inf. Process. Lett.*, 53(2):85–89, 1995.

[3] R. S. Barga et al. Consistent Streaming Through Time: A Vision for Event Stream processing. In *CIDR*, 363–374, 2007.

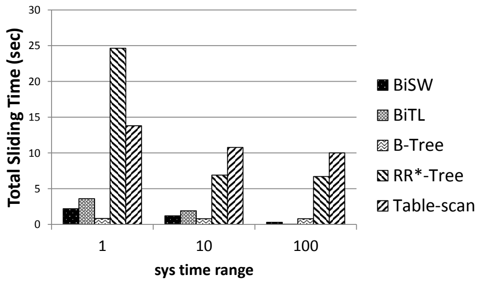
[4] N. Beckmann and B. Seeger. A Revised R*-Tree in Comparison with Related Index Structures. In *SIGMOD*, 799–812, 2009.

[5] R. Bliujute et al. R-Tree Based Indexing of Now-Relative Bitemporal Data. In *VLDB*, 345–356, 1998.

[6] R. Bliujute et al. Light-Weight Indexing of General Bitemporal Data. In *SSDBM*, 125–138, 2000.

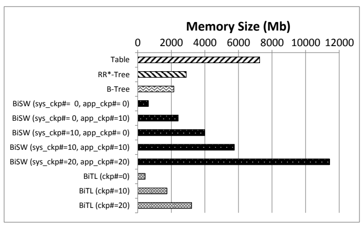


(a) Varying the number of application times

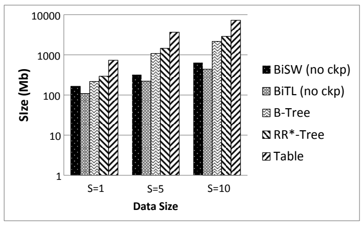


(b) Varying the number of system times

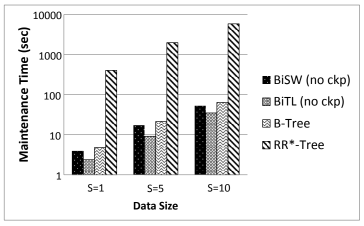
Figure 11: Experiment 5: Effect of the size of the system and application time domains.



(a) Space: Number of checkpoints

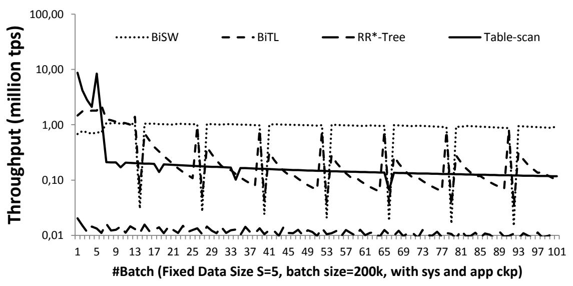


(b) Space: Data set size

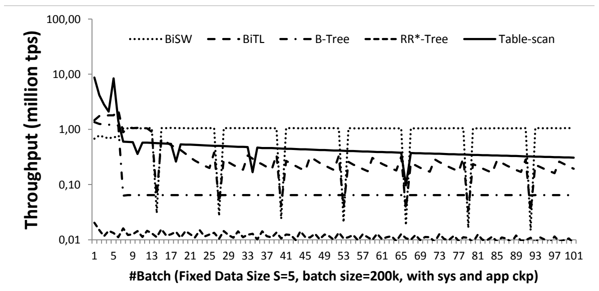


(c) Time: Data set size

Figure 12: Experiment 6: Indexing Cost



(a) Sliding both times



(b) Sliding system time only

Figure 13: Experiment 8: Continuous Queries

[7] H. Edelsbrunner. A New Approach to Rectangle Intersections Part I. *Int. Journal of Computer Mathematics*, 13(3-4):209–219, 1983.

[8] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The Time Index: An Access Structure for Temporal Data. In *VLDB*, 1–12, 1990.

[9] B. Gedik et al. Processing moving queries over moving objects using motion-adaptive indexes. *TKDE*, 18(5):651–668, 2006.

[10] L. Golab and T Joghnson. Data Stream Warehousing. In *ICDE*, 1290–1293, 2014.

[11] L. Golab and M. T. Özsu. *Data Stream Management*. Morgan & Claypool Publishers, 2010.

[12] M. Kaufmann et al. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *ICDE*, 471–482, 2015.

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

[14] M. Kaufmann et al. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 1173–1184, 2013.

[15] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *TODS*, 34(1):4, 2009.

[16] S. Krishnamurthy et al. Continuous Analytics over Discontinuous Streams. In *SIGMOD*, 1081–1092, 2010.

[17] K. G. Kulkarni and J.-E. Michels. Temporal Features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[18] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *TKDE*, 10(1):1–20, 1998.

[19] E. Liarou et al. Enhanced stream processing in a DBMS kernel. In *EDBT*, 501–512, 2013.

[20] M. A. Nascimento and M. H. Dunham. Indexing Valid Time Databases via B+-Trees. *TKDE*, 11(6):929–947, 1999.

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

[22] E. Ryvkina et al. Revision Processing in a Stream Processing Engine: A High-Level Design. In *ICDE*, 141, 2006.

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

[24] S. Saltenis and C. S. Jensen. Indexing of now-relative spatio-bitemporal data. *VLDB J.* 11(1):1–16, 2002.

[25] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31(2): 158-221, 1999.

[26] N. Shivakumar and H. Garcia-Molina. Wave-Indices: Indexing Evolving Databases. In *SIGMOD*, 381–392, 1997.

[27] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 79–96, 2001.

[28] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *PODS*, 263–274, 2004.

[29] Y. Tao and D. Papadias. Efficient Historical R-Trees. In *SSDBM*, 223–232, 2001.

[30] P. A. Tucker et al. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3):555–568, 2003.

[31] S. B. Zdonik et al. The Aurora and Medusa Projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.