Bitemporal Sliding Windows

by

Chang Ge

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The bitemporal data model associates two time intervals with each record - system time and application time - denoting the validity of the record from the perspective of the database and of the real world, respectively. One issue that has not yet been addressed is how to efficiently answer sliding window queries in this model. In this work, we propose and experimentally evaluate a main-memory index called BiSW that supports sliding windows on system time, application time, and both time attributes simultaneously. Our experimental results show that BiSW outperforms existing approaches in terms of space footprint, maintenance overhead and query performance.

Acknowledgements

First, I would like to thank my supervisor Prof. Lukasz Golab for his full support during my entire master's study period. Without his guidance, this work could not have been finished. Thanks goes to Prof. M. Tamer Özsu and Prof. Grant Weddell for their time to read this thesis.

I would also like to thank Dr. Anil K. Goel from SAP Waterloo and Martin Kaufmann from ETH Zürich for discussing the ideas, as well as for providing the machine and benchmark.

Dedication

To my parents and brother.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The *bitemporal* data model [23] [41] [22] denotes a relation with two time intervals from different perspectives: the system (or transaction) time interval is the time period during which a fact stored in the database is true, while the application (or business) time interval is the time period during which a fact is true in the real world. The validity of a fact may not be the same from different perspectives - for example, the account balance of $1000 for the year 2013 was inserted at year 2013, and at a later date in 2014 was corrected to $2000. At system time 2013, the database believed the balance to be $1000, but *in fact* (at application time year 2013), the balance should be $2000. Usually, system time and application time do not necessarily have to be the same. For example, consider a bitemporal database storing data about dinosaurs. The application time of these facts is millions of years ago, but the system time records when the facts were inserted into the database (for example, March 1st, 2014).

The bitemporal data model makes it possible to rewind the information to *what it actually was* in combination with *what was recorded* at some point in time, which enables the bitemporal model to provide both *historical* and *rollback* information. Historical information (e.g. "what was the year-2013 balance?") is answered by application time, and rollback (e.g. "in 2013, what was the balance recorded in the database?") is provided by system time.

The bitemporal data model is fundamentally required by numerous applications. One of those typical use cases is financial reporting, where information cannot be discarded even if it is erroneous. It is often desirable to be able to recreate an old report both as it actually looked at the time of creation and as it should have looked given corrections made to the data after its creation. Another use case is inventory management. An item's system-availability (for instance, 70) and application-availability (for instance, 100 on the shelf) can be different, due to 30 ordered but unshipped items. To generalize, any relation which distinguishes between system time and

application time falls into the category of the *bitemporal* data model.

The time attributes in the bitemporal data model naturally lead to *sliding windows*. Sliding windows provide a mechanism to discard old data over time, which are no longer of interest to users and/or must be deleted to satisfy data retention requirements. This intuitive approach is able to constrain analytics to a particular time interval, and is able to *slide* the focused interval by time. However, existing work on sliding windows focuses on system time only (e.g. data as it arrives to the system), without considering the equally important, but more flexible, application time dimension. In this work, we extend sliding window operations from exclusively on system time, to application time dimension and as well as both times.

1.1 Motivating Example

To emphasize the motivation behind and the difficulty of sliding windows upon the bitmeporal data model, let us first consider an intuitive example. For example, every day a network company receives transactions (e.g., to sign a new plan, to cancel an existing plan, etc.), and each transaction is associated with an application time interval (e.g. a plan has a start service date and an end service date). Also each transaction has a system time interval: the system start date is the date when the plan is signed, and the system end date records when the transaction is terminated, possibly because of cancellation. So a customer may sign a new 1-year plan contract on March 20th, 2014 (the system start time), which will become *activated* to provide service starting from April 1st, 2014 (the application start time), and will *expire* on April 1st, 2015 (the application end time). In this case, the system end time is infinity once the contract is signed. Later on, the customer cancels the plan on May 1st, 2014, and the system end time is set to be the cancellation date (May 1st, 2014). This story fits the bitemporal data model as system time and application time model different aspects of a plan contract.

We will formally define the bitemporal and sliding window model in Section 2; however, to make it simple, we assume that windows are defined by length of time (e.g., all the data that have arrived in the past 30 days) and a slide interval (e.g., slides every day). The network company is interested in the following sliding window queries:

- *Example 1: as of each day, how many contracts became signed in the past 60 days? (to slide on the system time)*
  This sliding window query measures the new signed contracts in past the 60 days, and is able to provide insights on sales performance. Logically, the contracts can be grouped by sign-in date, which implies grouping by transaction order. Physically, those logical-daily

buckets have identical order as the data arrival order, and are arranged in a sequence by arrival date (e.g., all data arrived at day $N+1$ are appended after the data at day N in the table, but before the position of data that arrived at day $N+2$). In fact this ordering reflects real-world streaming scenarios: data is natively clustered by system time, and system time has strict ascending order (e.g. later committed transactions are guaranteed to have larger system time).

As illustrated in Figure 1.1, at day 61, the logical bucket labelled 61 collects transactions which occurred on that day, leaving older partitions as is. A window spanning from day 1 to day 60 is applied to calculate the aggregation. Next day at 62, the window slides one bucket, and result for the new window instance is computed by adding the number of contracts in bucket labelled 61 to the old result computed at day 61, and deleting the number of contracts in bucket 1 from it.

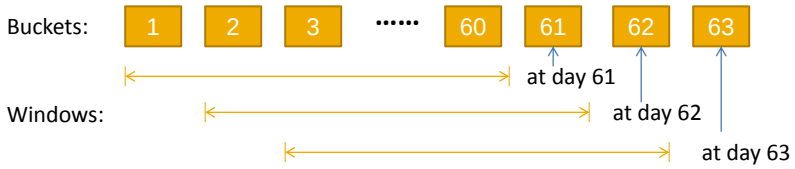


Figure 1.1: A Sliding Window for Example 1

- *Example 2: as of each day, how many contracts became activated in the past 60 days? (to slide on the application time)*
  At first glance, this query is identical to Example 1, except it slides on application time. One useful scenario for this query is to help calculate revenue, since new customers who started to use the service can be charged. The same logical layout pattern in Figure 1.1 can be applied here by labelling buckets using application time, and the windows slide on application time.

  However, Example 2 is quite different and difficult to slide compared with Example 1, in terms of concept and implementation. On the one hand, there is underlying *dependency*: the validity of application time loosely depends on system time, which means they are not fully orthogonal. For example, if a transaction is cancelled, it must not be considered in revenue. This implies that the validity of application time depends on the validity of system time, and system time has to be checked somehow even though the query slides on application time only. On the other hand, when new data arrive, it is possible that any logical bucket of the window incurs changes - indicating that the sliding window cannot be incrementally advanced and the buckets in the window have to be scanned again. Traditionally, there are two use cases in the area of data analytics. One type of data analytics

runs on static data, where the data is offline and accepts no more updates. The other type runs operations on online data, in which the data is able to accept insertion, deletion and updates. In practice, it is more desirable to be able to maintain application time sliding windows on the second use case, where tables can be updated and queries always run on the latest data.

For example in Figure 1.1, if the buckets represent system time (transaction order), then at each time, all new transactions go to the latest bucket only, and old buckets never get modified. In comparison, if the buckets denote application time (semantic order), then when new transactions are completed, their application times can be arbitrary: for instance, customers A and B may sign plan contracts on the same date (system time), but can specify different service start dates (application time). Hence transaction-order updates may result in modifying all semantic-order buckets in the worst case. Even worse for queries, the arbitrary updates disable the incremental computation of sliding windows, degrading sliding windows to recompute from scratch every time windows slide. For instance in Figure 1.1, after day 61 when the window is to slide, buckets 2 to 60 may have been changed at day 61, which makes the previous computed result at day 60 obsolete. Another case is that a contract valid at application time 30 may not still be application time valid at 31 if it is invalidated by some system time transactions.

- *Example 3: as of each day, how many contracts that were signed in the past 60 days, became activated in the past 30 days? (to slide on both times)*
  This query intends to obtain the number of contracts that were both signed in the past 60 days and activated in the past 30 days, and this query is practical for many purposes. In one example, this query is able to measure installation latency if cable setup is needed before providing Internet service. Also, this query is able to provide insights to evaluate the effect of advertisements and promotions (e.g. recent 30-day new signed contracts are eligible for some discounts if activations are within 30 days). Another example is to help user behaviour mining, such as analysing customer eagerness to use the services, by combining other customer features like age, gender, and so on.

  This query slides on both system time and application time as intervals, and the query results can be generated by intersecting the system time window and the application time window. The system time sliding (contracts were signed in the past 60 days) is entangled with application time sliding (contracts became activated in the past 30 days), which is a typical scenario of bitemporal sliding windows. As seen in the previous examples, sliding on one time intervals has many problems, not to mention the challenges of combining two dimensions.

  Note that an alternative way to answer this specific example is to maintain a *view* where the

differences (or subtractions) of two time intervals are stored. However, this view is neither effective nor efficient, not only because the subtraction values lose the information (e.g., start and end times) and the view maintenance incurs additional overhead, but also more importantly, system time and application time may not be directly comparable (recall the previous dinosaur example).

As inspired by the above examples, each type of query represents a typical sliding window case, involving temporal properties of two time dimensions. Sliding windows on the bitemporal data are not only very practical in the real world, but also very important for data analytics. In general, sliding windows provide a mechanism to constrain analytics to a particular time interval, and are able to *slide* the focused interval by time. For OLAP queries and analytic workloads, it is not uncommon to maintain more than one sliding window on different times and/or both dimensions within one bitemporal relation. However, existing temporal sliding windows work only focuses the first type (Example 1) of sliding system time only queries, but less work has been done to study the remaining equally important, but more challenging scenarios (Examples 2-3) of sliding application time and of sliding both times.

1.2 Challenges and Contributions

The examples point out challenges to implement sliding windows on the bitemporal data: the overlapping between the window intervals in sliding windows and temporal intervals in the bitemporal data model adds complexity to the sliding operations; the flexibilities of application time such as dependence on system time and arbitrary lifespan impose a burden on maintenance; and data strcutres to support sliding windows could potentially increases space overhead. In conclusion, the bitemporal sliding window is a challenging problem, and not yet well studied.

There exists a large amount of work on general temporal indexing structures, and extensive research on sliding windows has been conducted as well. Unfortunately, significantly less work has tried to combine both domains: using sliding windows on the (bi-)temporal table. A brute-force sequential scan of a bitemporal table whenever the window slides is feasible to answer queries; however, it is not acceptable due to latency. A better but still ineffective approach towards this problem is to support one domain natively, and to implement the other domain naively. For example, an alternative could build a general B-tree index on temporal attributes, and scan the B-tree to confine the window boundaries. However, as experiments show in Section 5, neither of these solves the problem of bitemporal sliding windows efficiently.

Therefore, the goal of this work is how to support bitemporal sliding windows efficiently,

with a focus on main memory data analytics and of real time business intelligence, such as in SAP HANA [15]. The contributions are as follows:

- This work extends the knowledge of the sliding window operations on the bitemporal data and studies the challenges of bitemporal sliding window queries.
- We propose an index structure called **BiSW** and two optimizations for particular use cases to effectively index bitemporal relations.
- Based on BiSW, query evaluation algorithms are designed to efficiently support three categories of sliding window operations on bitemporal relations.
- This work also experimentally analyses the performance and trade-off of different alternatives to support bitemporal sliding window queries, and results show that the BiSW outperforms the state-of-the-art Bitemporal Timeline Index and trees for most cases several times, in terms of space utilization, maintenance overhead, and query performance.

1.3 Organization

This thesis is structured as follows. Chapter 2 defines the bitemporal data model and the sliding window model, gives an expressive query syntax for sliding windows on the bitemporal relation, then defines the problem scope of this work, and summarizes the challenges. Chapter 3 overviews related work, which can be used to answer bitemporal sliding window operations. Chapter 4 proposes the structure of BiSW, explains the algorithms of three categories of query execution, and raises optimizations for two types of use cases. Chapter 5 compares the performance of different data structures that can be modified to support bitemporal sliding windows, and Chapter 6 concludes this thesis and provides insights for future work.

Chapter 2

Preliminaries

In this section, we give formal definitions of the bitemporal data model (Section 2.1) and the sliding window model (Section 2.2), and then discuss the problems (Section 2.3) associated with the bitemporal sliding windows, and finally summarize the challenges (Section 2.4).

2.1 Bitemporal Data Model

The bitemporal data model was originally introduced about twenty years ago [23] [41], but it had not attracted too much attention until it was included in the SQL:2011 Standard [31], after which major database vendors recently started to (partly) support (bi-)temporal relations. Unfortunately recent studies [24] [26] show that neither the research community nor the industry has made enough progress towards a native and full bitemporal operators support.

In this work, we model a bitemporal relation as a normal non-temporal relation with four additional time points, which constitute two time intervals: the application time interval represents the valid period of a tuple in the real world, and system time reflects the valid period in the database. The schema of a bitemporal relation can be interpreted as:

$$(\underline{key}, attributes, ..., \underline{StartApp, EndApp, StartSys, EndSys})$$

with the constrains that:

$$StartApp \leq EndApp$$
$$StartSys \leq EndSys$$

The primary key of the bitemporal table is composed of the non-temporal primary key (the $\underline{key}$) and the two time intervals. Each interval determines the validity of the *key* in one dimension, and both intervals together are able to uniquely determine a row. For the last four temporal attributes, they are time points/stamps that define the left-close right-open boundaries of the system and application time interval.

Visibility. A row is said to be *system visible* at system time t_{sys} when the following condition is met:

$$visible\ at\ t_{sys} \Leftrightarrow t_{sys} \in [StartSys, EndSys)$$

System time visibility is related to the rollback view (system snapshot) of the bitemporal table. Within a rollback view at system time t_{sys}, a row which is to be said *application visible* at t_{app} is defined as:

$$visible\ at\ t_{app} \Leftrightarrow t_{app} \in [StartApp, EndApp) \cap system\ time\ visible$$

The application time visibility is related to the historical view (application history) of the bitemporal table at a particular system-based snapshot. It does not make sense to measure application times alone without specifying the system time because different snapshots result in different historical views.

In general, system time and application time do not necessarily have the same scale, nor have any syntax correlations. Figure 2.1 shows a concrete bitemporal table (after system time 109 and application time 17) of a running example which will be used in the remaining sections.

In terms of the bitemporal table operations, system time is managed by the DBMS, and application time is treated as a user defined attribute and managed by the application itself. Insertion of a row is achieved by appending a new row into the table, setting StartSys to be equal to the system time point when insertion occurred, and EndSys to infinity. For example in Figure 2.1, row 8 is inserted at system time 107. Deletion of a row is fulfilled by setting the EndSys of that row to be the system time when deletion happened. For instance, row 7 is inserted at system time 107 and deleted at system time 109. Updating of non-temporal attributes or application time points never occurs in place; instead it is decomposed to a deletion of the old row and an insertion of a new one. For example at system time 102, the EndApp attribute of row 1 is to be updated, so row 1 is deleted, and row 2 is inserted with an updated EndApp attribute, and of course with a new pair of StartSys and EndSys.

Recalling the definition of visibility above, a straightforward way to interpret a bitemporal relation is to disintegrate the two time dimensions and to view it as a series of time slices for one dimension, with the other dimension changing over time. Figure 2.2 shows the example

ID	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	108
5	John	Costtown	100	13	14	105	108
6	John	Largevill	30	14	∞	105	106
7	John	Largevill	30	14	16	106	108
8	Max	Newtown	80	14	∞	107	∞
9	John	Largevill	30	11	12	108	∞
10	John	Newtown	120	12	15	108	∞
11	John	Largevill	30	15	16	108	∞
12	John	Largevill	50	16	17	109	∞

Figure 2.1: A Bitemporal Table

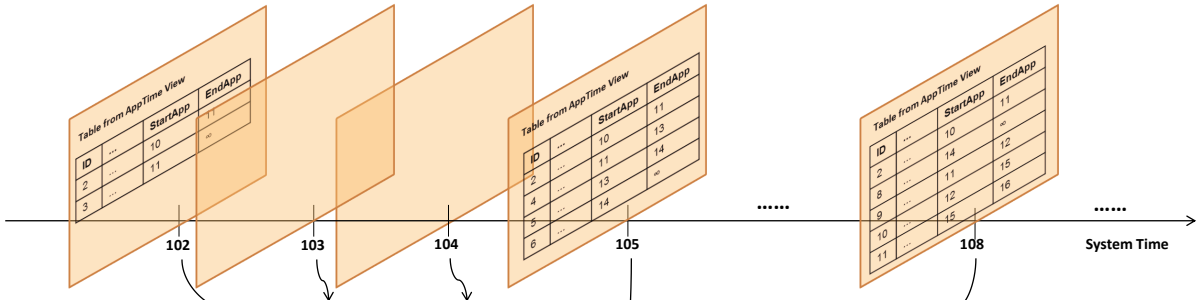
in Figure 2.1 by slicing the system time. At each system time point, the bitemporal table has a corresponding state, which exactly represents the physical evolution with system time. For example at system time 102, the bitemporal table contains 3 rows, but only the last 2 are visible. So at the second vertical application time dimension, the bitemporal table is *projected* to a single dimension temporal table including application time, and considers the row 2 and 3 only at the system time point. Projection at system time 105 and 108 are also exemplified in the figure. In this case, the layout grows by adding more sliced system times. On the other hand, a bitemporal relation can also be broken down into application time slices, as illustrated in Figure 2.3. Each application time has a projection (or view) of table: for example at application time 11, the bucket views the table as a list of transactions which make rows visible for application time 11. The application view grows longer over the system time.

2.2 Sliding Window Model

In this work, we refer to the *window* [17] as time-based span which is defined in terms of a time interval:

$$Window\colon [StartWin, EndWin]$$

The interval points $StartWin$ and $EndWin$ represent the two inclusive time boundaries for



Bitemporal Table at SysTime 102

ID	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	∞

Bitemporal Table at SysTime 105

ID	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	∞
5	John	Costtown	100	13	14	105	∞
6	John	Largevill	30	14	∞	105	∞

Bitemporal Table at SysTime 108

ID	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	108
5	John	Costtown	100	13	14	105	108
6	John	Largevill	30	14	∞	105	106
7	John	Largevill	30	14	16	106	108
8	Max	Newtown	80	14	∞	107	∞
9	John	Largevill	30	11	12	108	∞
10	John	Newtown	120	12	15	108	∞
11	John	Largevill	30	15	16	108	∞

Figure 2.2: Conceptual View by System Time

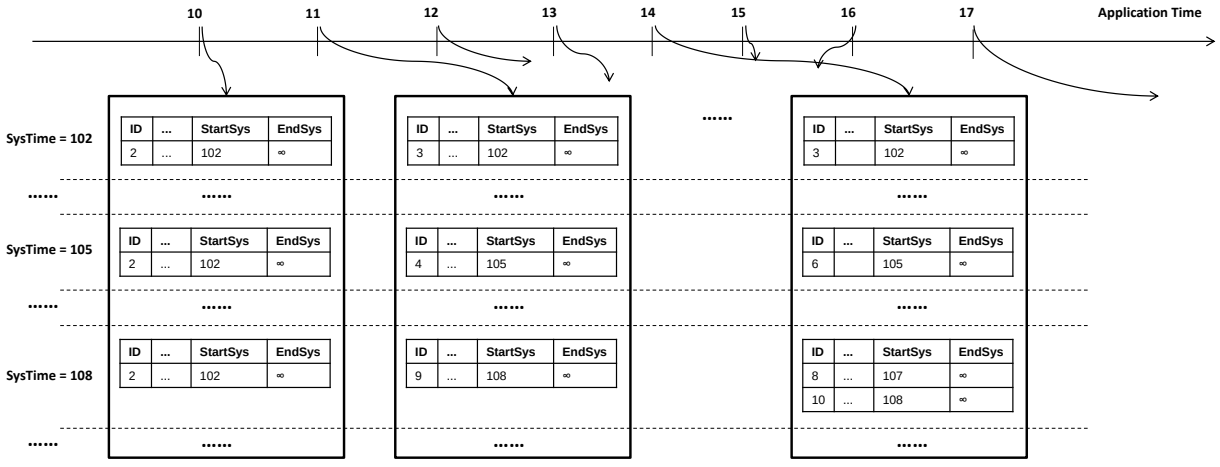


Figure 2.3: Conceptual View by Application Time

a window instance at a particular system time. Moving $StartWin$ and $EndWin$ (either forward or backward, or with different paces) creates a *sliding window*. In the following context for simplicity, we assume a fixed-length window, where the boundaries move at the same direction with the same speed.

Logically, tuples representing events at the same time (e.g., tuples arrive at same time, or tuples are visible at the same time) are grouped together into a bucket. Figure 2.4 shows an example of this concept. Each bucket is associated with a timestamp ts, where all tuples in that partition share this ts.

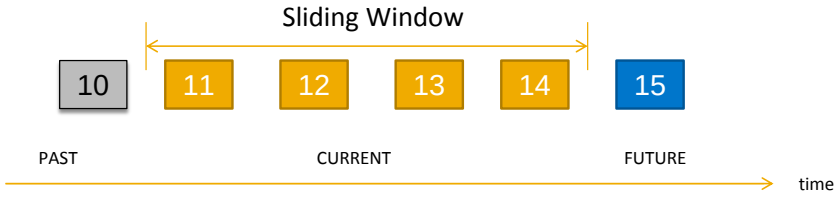


Figure 2.4: A Sliding Window Example

The *temporal sliding window* is a sliding window that slides on the temporal attributes of a (bi-)temporal relation. In this case, the data is logically partitioned by its temporal attributes, e.g., each partition holds tuples that become system visible at its timestamp. To clarify, if the underlying timestamp denotes application time, then we refer to this window as an application time sliding window, and similarly for the system time dimension.

Based on the temporal attributes involved in the query, we classify temporal sliding window queries into three classes: 1) sliding on the system time with the application time window fixed; 2) sliding on the application time with the system time window fixed; and 3) sliding on both times. Because there is no SQL standard to define such types of bitemporal sliding window queries, in this work we use the following pseudo-SQL template to express the classified three types of sliding window queries:

```
select expression()
from table
    [SYS, START sys_t, RANGE sys_len, SLIDE sys_pace]
    [APP, START app_t, RANGE app_len, SLIDE app_pace]
    [RATIO 1:N]
where predicates()
```

In the query statement, *expression()* can be one or more attribute names in the queried table *table*, or a function on attributes (e.g. $count(*)$, $sum(balance)$). *table* is the target bitemporal

table. The following two expressions define the sliding time dimension (*SYS* and *APP*), window start time point (*sys_t* as start system time point, *app_t* as the start application time point), fixed window range length as an interval (*sys_len* and *app_len*), and window sliding pace (*sys_pace* and *app_pace*). A pace of zero implies that there is no sliding on that time dimension. The keyword *RATIO* defines the frequency of sliding the system time window versus sliding the application time window, indicating a sliding loop pattern, in which first it slides the system time window once, and then it slides the application time *N* times. *predicates()* in the *where* statement can filter either temporal attributes (e.g. application starts before March 1st, 2014) or non-temporal attributes (e.g. customer has balance over 100). We will give concrete examples in Chapter 4.

2.3 Problem Statement

Following the above two sections, this work aims to index the bitemporal table effectively and to answer the bitemporal sliding window queries efficiently, with a focus on the following measurements:

- *Maintenance Overhead*. When the bitemporal table receives updates, the index structures should reflect the changes accordingly. Index maintenance overhead is measured as the time to update the index structure, and a shorter maintenance time is always desirable.
- *Query Performance*, which is measured as the total execution time. To be more specific, the query syntax in Section 2.2 contains two execution parts: the first part is to construct the initial windows, and the second part is to slide the windows. We measure these two steps separately. Faster execution time is better.
- *Memory Footprint*. It is obvious that smaller size is always desirable.

2.4 Challenges of The Bitemporal Sliding Windows

As mentioned earlier, there do exist naive alternatives to answer the bitemporal sliding window queries, such as materializing the conceptual views in Figure 2.2 and 2.3; however, the disadvantages are also obvious: large space overhead, heavy maintenance, and also no apparent algorithms for answering queries. The difficulties to evaluate the bitemporal sliding window queries fundamentally arise from three aspects: 1) semantic overlapping between the window interval(s) and the time intervals; 2) the unpredictability of application time; and 3) no efficient algorithms to evaluate queries. To be more specific, the challenges are discussed below.

- *life span versus life point*. Tuples in a non-temporal relation do not hold a time interval, and from the temporal view they have only a system time life point (e.g., arrival time, expire time). However, in the temporal relations, tuples are associated with life spans (either system, application, or both), which are time intervals and can overlap with the sliding window interval. For example, consider a sliding window in Figure 2.4. If the data is non-temporal and the time represents arrival time, when the window moves, the old bucket (labelled 10) can be expired. However if the data is temporal, a tuple whose life ends at 20 cannot simply be expired even if its bucket is beyond the window. This *overlapping* adds complexity to sliding.

- *unpredictable expiration*. Take the same example as above and assume partition labelled 10 holds tuples whose life span starts at 10. The end time of their life can be independently different. That implies the traditional *block evolution* [16] pattern does not work as individual tuples may have different life spans. In addition, when the application window slides, a tuple that is valid at application time app_t may not be still application time valid at $app\_t + 1$ even if its expiration time is much larger than $app\_t + 1$, due to the invalidation by some system time transactions.

- *arbitrary application time*. Take the same example as in Figure 2.4, and suppose it maintains a sliding window on application time. Application time is user defined, and has no ordering. New arrived data may result in insertion and/or deletion at any partition, make any previous computed results obsoleted, and disables the incremental computation (recall the mobile phone operator example in Section 1). However, for system time window, updates always occur at latest partition only.

- *system and application time are inter-dependent*. System time and application time are not fully independent. On the one hand, updating an application time requires a new system version, but the opposite is not necessarily true. In Figure 2.1, the updating application time in row 1 incurs *cascading* updates on system time: EndSys in row 1 has to be updated in order to fulfill this operation. On the other hand, sliding on system time may validate and invalidate items in application time. For instance, row 1 was visible for application time 20 at system time 101, but after system time slides to 102, at application time 20, this row becomes invisible due to this cascading invalidation. Furthermore, system time has strictly ascending order which is guaranteed by the DBMS (e.g. later arrived data is guaranteed a larger system timestamp), but for application time, there is no constraint of ordering.

In conclusion, sliding windows on temporal data is a different but more challenging topic compared with traditional non-temporal windows.

Chapter 3

Related Work

There are not many related works directly addressing the context of temporal sliding windows, but there exist several possible approaches which can be adapted to the bitemporal sliding window problem. In this chapter, we first overview several structures that can be used to index (bi-)temporal relations in Section 3.1, then we introduce the latest research work that is designed to support general temporal queries in Sections 3.2 and 3.3. Section 3.4 reviews related work on sliding windows, and Section 3.5 also surveys techniques deployed in current commercial database systems.

3.1 General Indexing on (Bi-)Temporal Attributes

The B-tree [6] and its variants (e.g. B+-tree [12], B*-tree [28]) implementations for temporal attributes differ on not only the native design trade-offs between internal and leaf nodes among various B-tree structures, but also on the representation of keys at each leaf node. For the key in the tree nodes, it can be either the non-temporal key without considering temporal attributes, or a time point (e.g. boundary of intervals [13]), or a composited value (e.g. MAP21 [33]) to encode a time interval.

The Time Index [14] uses a B+-tree to index valid time points, and stores all the visible rows identifiers at the beginning of each leaf node, followed by all changes. The disadvantage of the Time Index is the identifiers are copied to each version as long as a tuple is valid, and they consume considerable space. The Interval B-tree (IBT) [4] indexes tuples based on the boundary time point which is stored in the node as the key, and tuples with same time point are chained together. The nested tree-list imposes considerable maintenance overhead (updating and deleting

data to and from IBT) and searching overhead for non-temporal attributes. MAP21 [33] provides an efficient way to index ranges by mapping a time range to an unique value and indexing the unique value as keys in a B+-tree. But this mapping incurs additional computation complexity.

The R-tree [19] was originally designed to index spatial data, which usually contains two or more dimensions. In contrast to the B-tree, the (multi-dimensional) R-tree and its derivatives (R+-tree [38], R*-tree [8]) provide one more dimension to index keys such that both key and time can be indexed together. In the R-tree, objects are modelled by means of minimum bounding rectangle (MBR), which are stored as entries in a leaf-level node, together with pointers to the data tuples containing those MBR. All variants of the R-tree try to minimize the MBR and unoccupied space in MBR in order to reduce IO and search branching complexity, by trading off other factors. For example, the R+-tree [38] duplicates objects to avoid node overlapping; the R*-tree [8] allows deletion and reinsertion of the same entries to find them better places (smaller MBR and unoccupied space), considering the fact that R-tree structures are highly susceptible to the order in which their entries are inserted. Th Revised R*-tree (RR*-tree) [9] optimizes the insertion imbalance in the R*-tree.

In particular, some R-tree variants are designed to meet specific temporal indexing requirements. For example, the Historical R-tree [43] maintains a R-Tree for each timestamp to efficiently answer time point queries, but it requires large space overhead. The 2R-tree [32] uses two R-trees, in which one tree stores current system time valid tuples to answer now-valid queries, and system time invalid tuples are moved in the second R-tree, which costs additional overhead.

There exists intensive research on general uni-temporal indexing which indexes one time dimension (either application or system time); however, significantly less research has been done on indexing bitemporal data. For bitemporal indexing, one straightforward way is to use a spatial index structure to index both dimensions at the same time, due to the similarity between spatial and bitemporal data: a bitemporal tuple can be represented as a rectangle, which is bounded by its application and system time intervals. A two-dimensional R-tree can be used to index bitemporal times such as proposed by the GR-tree [10] and the 4R-tree [11], while a multi-dimensional R-tree can add more dimensions to facilitate non-temporal attribute access. This intuitive approach is able to reuse the existing R-tree operations, but at the same time also inherits the overhead of it.

On the other hand, these two dimensions can be fully-independently indexed. In theory, any two uni-temporal index structures discussed above can be combined to support bitemporal indexing. But the disadvantages are obvious: the two independent structures not only consume more space, but also penalize query performance. As a baseline, we will describe the query evaluation and experiment results using two B-trees to index two times separately.

In addition to trees, non-tree structures are also able to provide support for bitemporal data.

Differential files [21] storing changes occurred incrementally in a log resemble one intuitive approach to index tuples by system time. Furthermore, multi-version techniques can be applied to trees to enable restoring trees at different version in time, such as multi-version B-tree (MVBT) [7] and multi-version 3D R-tree (MV3R) [44]. But this technique requires large space as the version increases.

In conclusion, all the structures mentioned above are designed to index general temporal tables, with consideration on reducing disk IO. However, none of them is specifically designed to support temporal sliding windows, and as experiments show later, they are not sufficient to answer the queries.

3.2 Timeline Index

One of the recent works is the Timeline Index [27], and as it is used as a building block in the Bitemporal Timelime Index (Section 3.3) and in our proposed BiSW structure as well. The idea of the Timeline Index is to keep track of all visible rows from the temporal table at every point of time. This is achieved by recording the RowIDs of invalidation (e.g. deletions) and validation (i.e. insertions) for each version, in version order. By scanning this information, operators can determine changes between versions and/or establish the set of active tuples for a specific version.

Figure 3.1 gives an example of a system time temporal table, and its corresponding system timeline index is illustrated in Figure 3.2. As shown in Figure 3.2, the *Event List* keeps track of each validation and invalidation event. Validation events are marked with a "1" and invalidation events are marked with a "0". The events in Event List are sorted by system time as the event occurred; i.e. row 1 is validated before row 2. The order of events created at the same system time is undefined.

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

Figure 3.1: An Simplified System Table

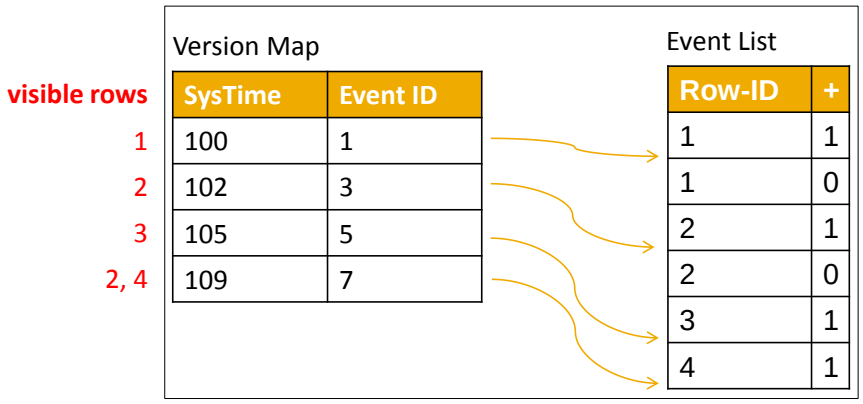


Figure 3.2: System Timeline Index (Physical Implementation)

visible rows	SysTime	Event ID
1	100	+1
2	102	-1+2
3	105	-2, +3
2, 4	109	+4

Figure 3.3: System Timeline Index (Logical View)

The *Version Map* keeps track of the list of events that are seen by each system time. This is achieved by storing the end offset for each system time in the Event ID column. By concurrently scanning and merging the Version Map and Event List, it is possible to reconstruct all the visible rows of the temporal table. Figure 3.2 shows the visible rows for each system time in red, but this information is implicit and not materialized. In Figure 3.2 at system time 100, row 1 becomes visible, so its ID is toggled to be visible (the first row in the event list). This validation event is stored in the Version Map by adding the offset of this event to indicate the end of events which occur at system time 100. Corresponding to its logical view shown in Figure 3.3, the activation of RowID 1 is represented as $+1$ at the first row. At system time 102, row 1 becomes invisible, and row 2 starts to be visible, so these two events are recorded in the event list, and the version map is updated for the event ID. In Figure 3.3, these two events are represented as $-1 + 2$ at the second row. Figure 3.3 is equivalent to Figure 3.2 and more human-friendly to read. In the rest of this work, we will describe the Timeline Index using its logical view.

In order to utilize compression and to facilitate scans, both the version map and event list

are physically implemented as arrays. For maintenance, when new data arrives at system time t, the new information can be appended to the end of Version Map and Event List, because t is guaranteed to be larger than the largest existing system time. In practice, the timeline index is useful only to index system time, because system time has strict ascending ordering, therefore later arriving data can appended to the end of event list and version map.

3.3 Bitemporal Timeline Index

In order to enhance the Timeline Index to support application time, the Bitemporal Timeline Index (abbr. BiTL) [26] was proposed recently. The methodology of BiTL focuses on two aspects. The first one is *lazy materialization*. When data arrives, BiTL only maintains a timeline index on the system time dimension, which is exactly as it is in the Timeline Index above. Later when an application time related query is issued, it will construct a timeline index for application time. Thus BiTL postpones the materialization to query runtime, which enables fast maintenance when data arrives. A system timeline index is maintained over time, but the application timeline index (if any) is not maintained when data arrives.

The other feature of BiTL is to adopt the idea of *conceptual view* by system time (in Figure 2.2), where conceptually at the first level a bitemporal relation is clustered by system time only, and for each system time t, at the second level only those tuples which are visible for system time t are considered in the application timeline index. Compared with naive structures which independently index the two times, the BiTL makes use of the coupling property. For example in Figure 2.1, at the end of system time 101, row 1 is visible from the perspective of system time, therefore the application timeline index at system time 101 should cover row 1; however, at the end of system time 103, row 1 is invisible from the view of system time, hence the application timeline index at system time 103 should not include row 1 in its Version Map or Event List. In comparison, under the methodology of independently indexing two times, after system time 100 (row 1 insertion time), row 1 always exists in the indexing structure on application time, indicating that it starts at application time 10 (without considering its invalidation at system time 102). This is one of the major reasons why tree indices are not efficient. In BiTL, the late-materialized application timeline index is associated with a system time, and only contains visible tuples at that system time.

Take the running example in Figure 2.1. Between system time 101 and 106, data arrives, and only the system timeline index is updated. Assume that at the end of system time 106, an application timeline index is created due to some query requests. Figure 3.4 shows the application timeline index, and the bitemporal table at system time 106. The timeline index on application time looks similar to the timeline index on the system time in Figure 3.2, except it indexes on

AppTime	Event ID
10	+2
11	-2, +4
13	-4, +5
14	-5, +7
16	-7

ID	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	∞
5	John	Costtown	100	13	14	105	∞
6	John	Largevill	30	14	∞	105	106
7	John	Largevill	30	14	16	106	∞

Figure 3.4: Application Timeline Index (at system time 106)

the application time. Between system time 107 to 109, data continues arriving and only the system timeline index is maintained, and at the end of system time 109, the bitemproal table and system timeline index is illustrated in Figure 3.5. Now supposing some queries have predicates on system time 109 and application time (for example, 15), a *delta* application timeline index, which holds changes since last application timeline, is triggered to be created as shown in Figure 3.6, taking the bitemporal table and system timeline between 107 to 109 (Figure 3.5) as input. Depending on the query types (described in Section 4), the delta can be consumed directly, or merged with previous application timeline index at system time 106 to a new application timeline index at system time 109 (as shown in Figure 3.7).

Regarding the BiTL methodologies above, the application timeline index at system time 109 is not created until needed. Furthermore, row 1 for instance, is not indexed because it is invisible at system time 109. Compared with independently indexed application timeline where all rows should be indexed, this loose-coupled application timeline index in Figure 3.7 is much smaller.

Another major reason that trees are not as efficient as BiTL is that the Timeline Index and BiTL are used in memory, while trees were designed for disk. Experiments show that BiTL outperforms tree-based structures which were designed to optimize IO, by several orders of magnitude for ad-hoc bitemporal time travel (restoring the table to a previous version), aggregation and join queries.

However, BiTL is not suitable for bitemporal sliding window queries. *Whenever system time slides, BiTL requires the runtime creation of the application timeline index, which increases the query response time*. As experiments demonstrate, those creations take considerable time. Furthermore, BiTL maintains one system timeline index and probably more than one application timeline index, where a RowID may be duplicated in the system timeline index and in the application time index; thus *BiTL results in inefficient space utilization*. It is often desirable to achieve an efficient space usage and a fast query response time.

ID	Name	City	Balance	StartApp	EndApp	StartSys	EndSys
4	John	Largevill	30	11	13	105	108
5	John	Costtown	100	13	14	105	108
7	John	Largevill	30	14	16	106	108
8	Max	Newtown	80	14	∞	107	∞
9	John	Largevill	30	11	12	108	∞
10	John	Newtown	120	12	15	108	∞
11	John	Largevill	30	15	16	108	∞
12	John	Largevill	50	16	17	109	∞

SysTime	Event ID
...	...
107	+8
108	-4, -5, -7, +9, +10, +11
109	+12

Figure 3.5: Partial System Timeline Index ($sys \in [107, 109]$)

3.4 System Time Sliding Widow Processing

The data structures in previous sections support general temporal indexing; however, it is not clear how bitemporal sliding window operators can be answered using these structures.

Most existing work on temporal sliding windows focuses on the logical query operator and adds SQL syntax to facilitate query expression, such as described in [16] [29] [37] [45] [5]. However, there are less related works on how to physically implement a temporal sliding window, not to mention how to implement a bitemporal sliding window. The Wave Index [40] proposes several algorithms to facilitate index update on the system time sliding window, with an emphasis on minimizing disk IO. The Doubly Partitioned Index [18] categorizes each temporal tuple into a partition, determined by insertion time range and expiration time rage, but it considers only one time dimension with pre-defined ranges; and due to potentially large combinations of insert/expire ranges, the Doubly Partitioned Index may incur large space overhead.

Data Stream Management System (DSMS) [17] is the type of system that ingests unbounded streaming as input and answers continuous queries. DSMS supports sliding window operations natively, but usually it relies on a global system clock in order to ensure a convenient and well-

AppTime	Event ID
11	-4, +9
12	-9, +10
13	+4, -5
14	+8, +5, -7
15	-10, +11
16	+7, -11, +12
17	-12

Figure 3.6: Application Timeline Index Delta ($sys \in [107, 109]$)

AppTime	Event ID
10	+2
11	-2, +9
12	-9, +10
14	+8
15	-10, +11
16	-11, +12
17	-12

Figure 3.7: Application Timeline Index (at system time 109)

behaved notion of time. In practical DSMS scenarios such as the distributed sensor network, the incoming stream does not strictly follow system time order (e.g., because of network latency). Therefore some techniques such as the overflow chain [30] and slack buffer [47] are designed to tolerate the out-of-order data and to correct query results. But these techniques consider system time only and regard those challenges such as out-of-ordering as exceptions, not an common cases, which is not true for application time. Some works have tried to extend the DSMS to represent application time, such as heuristic heartbeat [42], punctuation [46] and revision [35], but they didn't treat application time equally as first class citizen.

3.5 Bitemporal Support in Commercial DBMS

IBM DB2 [36] extends SQL to include uni-temporal and bitemporal functionality, but we are not aware of how this feature is implemented. Oracle introduces Flashback Data Archive [34] to store data changes using background processes, which in turn can be treated as an unitemporal system time index. PostgreSQL GiST [20] supports the R-tree index, which depends on users to implement the indexing for (bi-)temporal data.

Teradata publishes its temporal query processing in [3], but its implementation is through functional query rewrites to convert a temporal query to a semantically-equivalent non-temporal counterpart, by adding time-based constraints. This SQL-level query rewriting incurs burden on

the query optimizer, and also performance degrades when compared with kernel-level temporal structure implementation.

Current release of the SAP HANA [15] can support temporal operators on system time, and it does not support the application time as efficiently as the system time.

Again, these commercial systems supports general temporal queries, but they do not support sliding window queries.

Chapter 4

The Bitemporal Sliding Window

In this section, we will first propose the BiSW structure (Section 4.1), and then describe how to evaluate queries using BiSW (Sections 4.2 to 4.4). To further improve performance, we propose an improved version of BiSW in Section 4.5, and checkpointing in Section 4.6.

4.1 The BiSW Index

Given the challenges of bitemporal sliding windows and the performance advantages of the timeline index, we aim to provide a good tradeoff between 1) update cost, 2) query cost, and 3) space cost. As described below, the BiSW index overcomes the overlapping between sliding windows and temporal intervals, and efficiently indexes two time dimensions to enable fast and flexible window sliding.

In order to better understand the philosophy behind the BiSW, Figure 4.1 revisits the two closely related works at a high level: Timeline Index and the BiTL. The Timeline Index does not index application time, and therefore does not directly support sliding windows on application time or both times. BiTL maintains a system timeline index over time, and lazily materializes the application timeline index at particular system time points. As outlined earlier, BiTL does not efficiently support sliding windows involving application time: for bitemporal sliding windows, whenever system time window slides, BiTL has to construct a new application timeline index; and even for queries sliding application time only, the application timeline index materialization upfront cost can be prohibitive.

As a comparison, Figure 4.2 illustrates the high level organization of BiSW. The main differences are highlighted as the following:

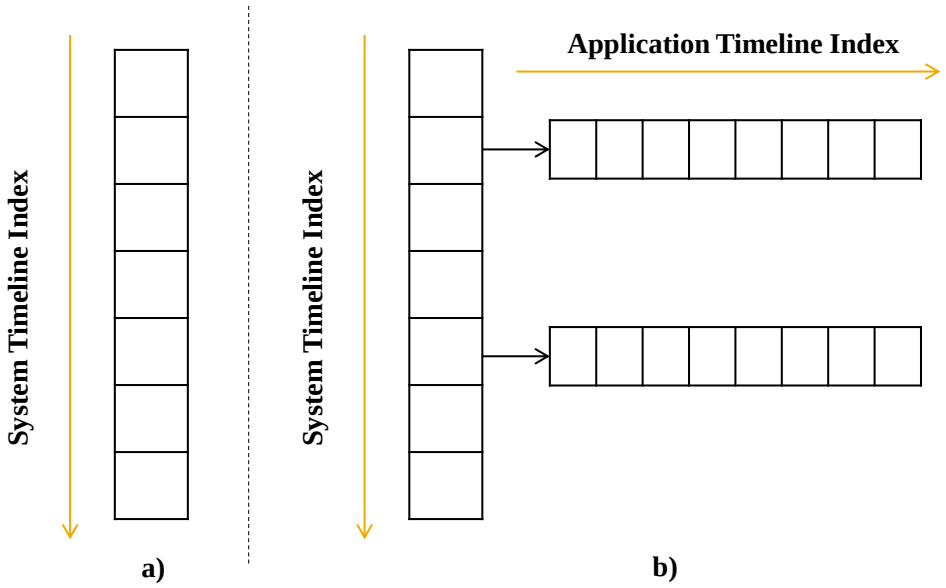


Figure 4.1: Overview of a) Timeline Index and b) BiTL

- Application time partitioned system timeline index. Rather than ignoring the application time in the Timeline Index or materializing application timeline indices per system time as in BiTL, BiSW maintains a system timeline index per application time point (or interval range). In this case, BiSW represents information for both the system time and application time.

- Non-materialized application timeline index. Since BiSW differentiates the application time from the system time, it does not have to compute the application timeline index for a particular system time. The horizontal dashed rectangles represent the corresponding application timeline index at particular system times. The application timeline index for a particular system time resides at the same position in different system timeline indices.

- Incremental computation for window sliding. BiSW is always evaluating sliding window queries in an incremental way. Compared with BiTL where a new application timeline index has to be recomputed and rescanned whenever system time slides, an incremental computation for sliding windows benefits query performance.

Conceptually, we describe the BiSW in a 2-dimensional space as depicted in Figure 4.3. The horizontal axis represents the application time, and vertical axis shows the system time. The key idea of BiSW is *an application time partitioned system timeline index*. Each application time is

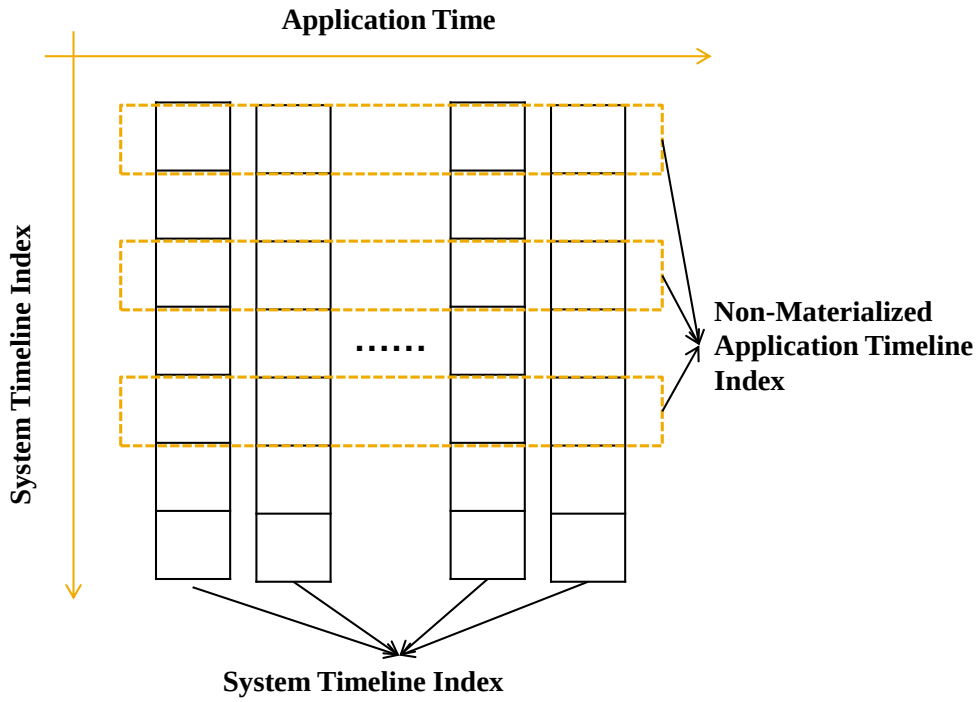


Figure 4.2: Overview of BiSW

associated with a *bucket*, which is implemented as a system timeline index described in Section 3.2. A bucket timestamped with application time app_t is created to hold all events that occur at application time app_t. A *cell* with coordinates (x, y) holds those events which occurred at application time x and at system time y. Note that Figure 4.3 has empty cells conceptually; however, the system timeline indexes are implemented as arrays without empty cells, as shown in Figure 3.2.

For index maintenance under this organization, BiSW utilizes the same append-only property of ascending system time. When new data arrives following the system time order, the data is partitioned on-the-fly by their application time, and appended to the end of the buckets according to their application times. For the updates happening at system time sys_t, visually only those horizontal cells which share the same system time coordinate sys_t will be touched. Figure 4.3 shows an example of updating at system time 105, at which the bitemporal table looked like in Figure 4.4 (only showing relevant rows at system time 105). For instance, the row whose ID is 4 and application time interval starts at 11 and ends at 13, becomes valid at system time 105, therefore this validation event is reflected in the cell with coordinates $(11, 105)$ (+4), and in the cell with coordinates $(13, 105)$ (-4). Note that all events occurring at system time 105 will only be

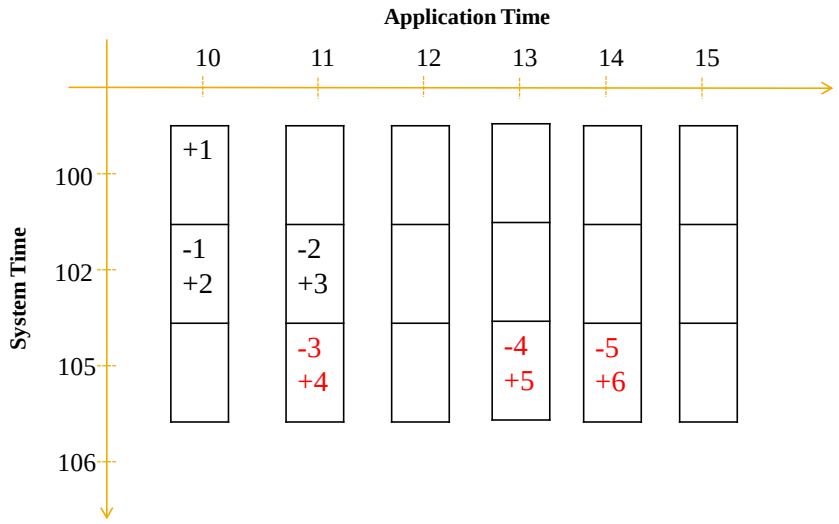


Figure 4.3: BiSW Updating (at system time 105)

ID	Name	City	Balance	StartApp	EndApp	StartSys	EndSys
3	John	Largevill	40	11	∞	102	105
4	John	Largevill	30	11	13	105	∞
5	John	Costtown	100	13	14	105	∞
6	John	Largevill	30	14	∞	105	∞

Figure 4.4: Bitemporal Table (at system time 105)

reflected in the cells with system time coordinates $(*, 105)$, which means at system time 105, the events are appended at the end of the buckets. In general, for maintenance from the point of each bucket, if standing at each application time point, update operations append information to the bucket end only. In terms of the conceptual views in Section 2.1, BiSW matches the conceptual view by application time (Figure 2.3).

We use the term *delta* to refer to a fraction of cells in BiSW throughout this paper. Delta is the logical unit not only for maintenance, but also for queries explained later. For the maintenance at certain system time sys_t, BiSW only touches a *horizontal delta* at sys_t. In addition, a *vertical delta* at application time app_t covers events that occurred at app_t. Therefore, sliding window queries are able to get benefits from this delta concept. For point lookup at a particular time, there can be an additional index to facilitate position locating, or basic binary search will suffice.

Compared with state-of-the-art BiTL, which clusters the tuples by system time and postpones

any construction of application index to runtime, the BiSW structure partitions the table by application times on-the-fly when new data arrives, and organizes indices to cluster by application time. Under this structure, one of the major benefits is that BiSW decouples the application times with system times: it is able to support queries which can fetch at any system time, or at any application time, or at both times.

Another major advantage of BiSW over BiTL is about half of the total space is saved. BiTL materializes one system timeline index plus one or more application timeline indices (Figure 4.1 materializes two application timeline indices), which indicates that the same row ID may appear in the system timeline index, as well as in one or more application timeline index as long as they are still application time visible. In comparison, BiSW does not double-materialize, therefore BiSW significantly outperforms BiTL in terms of space footprint.

4.2 Query Type 1: Slide System Time

As classified in Section 2.2, the first type of sliding window query is to slide on system time, with the application time window fixed. One instance of the sliding system time queries on the bitemporal table in Figure 2.1 is:

```
select *
from table
    [SYS, START 100, RANGE 4, SLIDE 1]
    [APP, START 10,  RANGE 4, SLIDE 0]
    [RATIO 1:0]
```

In the query, the statement specifies that the initial system time window starts at system time 100 with a range of 4, indicating that the initial system time window covers $[100, 104]$. Similarly, the initial application time window spans application time $[10, 14]$. The *SLIDE* keyword defines a pace that each sliding takes (e.g., the system time window advances 1 time per sliding, and the application time window does no sliding). Taking Figure 2.1 as a running example, the corresponding BiSW is shown in Figure 4.5, and the execution sequence for the above query is listed in Table 4.1. Algorithm 1 describes the query evaluation process.

The first step is to construct the initial windows. After getting all row identifiers which are valid in the rectangle, the user defined function $expression()$ (here it is $*$) can be called back to proceed. Retrieving other columns using row identifiers is out of the problem scope of the BiSW.

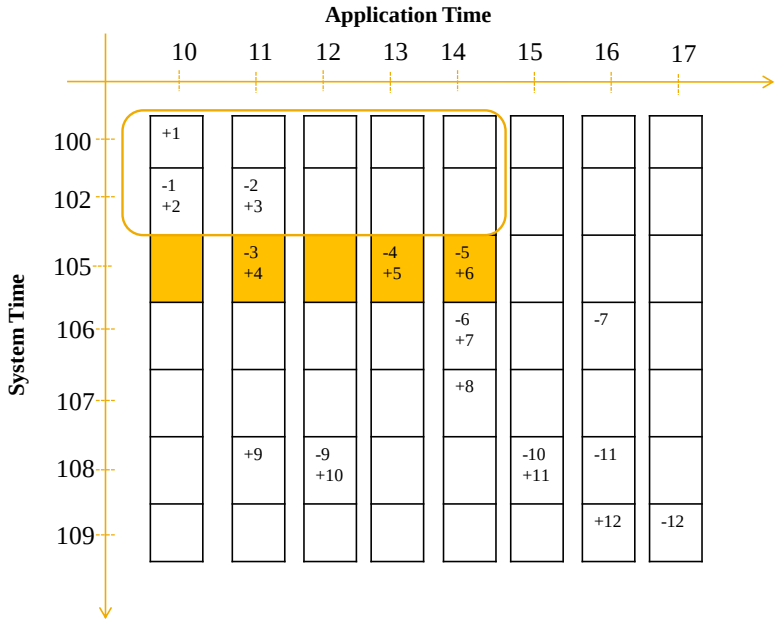


Figure 4.5: Slide System Time

After the initial windows have been initialized, the second step is to incrementally compute deltas for the system time sliding window (line 6). In Figure 4.5, this delta is represented as a *horizontal* rectangle (dark part), with the fixed application window as its length, and one system time as its width because of sliding 1 pace. Depending on the select $expression()$, the $expression()$ can utilize the delta directly (e.g. select sum can be incrementally computed by adding/subtracting values from deltas), or consume the updated visible rows after merging with existing visible IDs (e.g., selecting $topK$ has to be filtered from the whole ID set).

Then repeatedly constructing the deltas until *SLIDE* loop (here it is 4) is achieved. Note that the BiSW returns row IDs for every window instance, and it relies on the DBMS how to fetch real data (e.g. random access, batch scan and etc.), which is independent and out of our problem scope.

4.3 Query Type 2: Slide Application Time

The second type of the sliding window query is to slide on the application time and to fix the system time interval. An instance of this type of query can be the following:

Algorithm 1 Query Evaluation for Sliding System Time

Require:
sys_start_all: overall start system time
app_start_all: overall start application time

1: **procedure** GET START CONDITION
2: $valid\_IDs \leftarrow empty$ ▷ empty set for all IDs
3: $valid\_IDs \leftarrow getEventFromRECT($
$sys\_start\_all, sys\_end, app\_start\_all, app\_end)$
▷ rectangle starts from origin till sys_end and app_end
4: $expression(valid\_IDs)$ ▷ run user defined function
5: **end procedure**

6: **procedure** INCREMENTAL SLIDE SYSTEM
7: $delta\_IDs \leftarrow empty$ ▷ empty set for delta IDs
8: $i \leftarrow 1$
9: **while** $i <= slide\_count$ **do**
10: $crnt\_sys \leftarrow sys\_end + i$
11: $delta\_IDs \leftarrow getEventFromRECT($
$crnt\_sys, crnt\_sys, app\_start\_all, app\_end)$
▷ get delta IDs at $crnt\_sys$
12: merge $delta\_IDs$ into $valid\_IDs$
▷ update all IDs if $expression()$ needs
13: call $expression()$ ▷ run user defined function
14: $i \leftarrow i + 1$
15: **end while**
16: **end procedure**

Table 4.1: Execution Sequence for Sliding System Time

Seq#	AlgLn#	Event	DeltaRows	ValidRows
0	3	initial windows sys: [100,104] app: [10,14]	+3	3
1	11	slide sys at 105 sys: [105,105) app: [10,14]	-3+6	6
2	11	slide sys at 106 sys: [106,106) app: [10,14]	-6+7	7
3	11	slide sys at 107 sys: [107,107) app: [10,14]	+8	7,8
4	11	slide sys at 108 sys: [108,108) app: [10,14]	+10	7,8,10
5	—	—	—	—

```
select *
from table
    [SYS, START 100, RANGE 5, SLIDE 0]
    [APP, START 10,  RANGE 2, SLIDE 1]
    [RATIO 0:1]
```

Similarly using the same running example, Figure 4.6 shows the BiSW structure, and Table 4.2 illustrates the execution sequence. Algorithm 2 describes the query answer process.

The main syntax difference with previous query is in the sliding window part. This query intends to have a non-sliding system time window starting from system time 100 and ending at 105, and to have an initial application time window spanning $[10, 12]$ and to slide application time 1 pace. For the algorithm, the difference is reflected in line 8, which constructs the delta as a *vertical* rectangle (dark part in Figure 4.6).

Algorithm 2 Query Evaluation for Sliding Application Time

Require:
sys_start_all: overall start system time
app_start_all: overall start application time

1: **procedure** GET START CONDITION
same as in Algorithm 1.
2: **end procedure**

3: **procedure** INCREMENTAL SLIDE APPLICATION
4: $delta\_IDs \leftarrow empty$ ▷ empty set for delta IDs
5: $i \leftarrow 1$
6: **while** $i <= slide\_count$ **do**
7: $crnt\_app \leftarrow app\_end + i$
8: $delta\_IDs \leftarrow getEventFromRECT($
$sys\_start\_all, sys\_end, crnt\_app, crnt\_app)$
▷ get delta IDs at $crnt\_app$
9: merge $delta\_IDs$ into $visible\_IDs$
▷ update all IDs if $expression()$ needs
10: call $expression()$ ▷ run user defined function
11: $i \leftarrow i + 1$
12: **end while**
13: **end procedure**

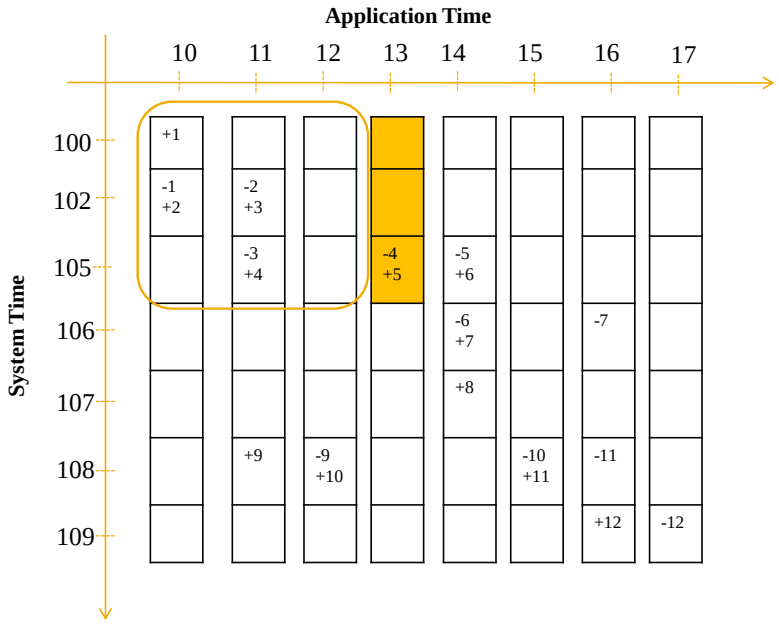


Figure 4.6: Slide Application Time

Table 4.2: Execution Sequence for Sliding Application Time

Seq#	AlgLn#	Event	DeltaRows	ValidRows
0	1	initial windows sys: [100, 105] app: [10, 12]	+4	4
1	8	slide app at 13 sys: [100, 105] app: [13, 13)	-4+5	5
2	8	slide app at 14 sys: [100, 105] app: [14, 14)	-5+6	6
3	8	slide app at 15 sys: [100, 105] app: [15, 15)	$\emptyset$	6
4	—	—	—	—

4.4 Query Type 3: Slide Both Times

The third type of sliding window query is to slide the system time and the application time. The following example is used to explain the idea:

```
select *
from table
    [SYS, START 100, RANGE 4, SLIDE 1]
    [APP, START 10,  RANGE 2, SLIDE 1]
    [RATIO 1:2]
```

In this type of query where both the system and application time windows slide, the example query has an initial system time window of $[100, 104]$ and an initial application time window of $[10, 12]$. The keyword *RATIO* is used to control the percentage of sliding system versus sliding application time. The ratio $1 : 2$ defines an execution sequence, which is constituted by loops of sliding system time window once and sliding application time window twice afterwards. Figure 4.7 shows an instance of BiSW on the same bitemporal table.

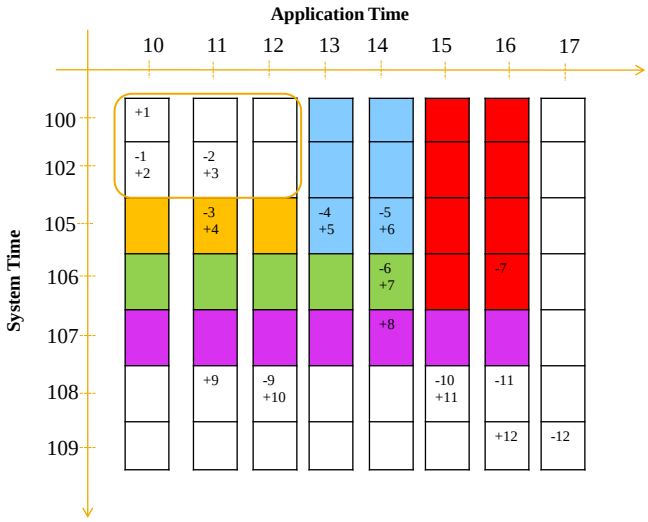


Figure 4.7: Sliding Both Times

The query plan also consists of two parts: the first step is to answer the initial windows, and the second step is to compute deltas incrementally. Algorithm 3 explains the execution process. Line 10 is to get the system time delta (which is a horizontal rectangle similar to Query Type

Algorithm 3 Query Evaluation for Sliding Both Times

Require:
sys_start_all: overall start system time
app_start_all: overall start application time

1: **procedure** GET START CONDITION
same as in Algorithm 1.
2: **end procedure**

3: **procedure** INCREMENTAL SLIDE APPLICATION
4: $ID\_sys\_delta \leftarrow empty$
5: $ID\_app\_delta \leftarrow empty$
6: $i \leftarrow 1$
7: **while** $i <= slide\_count$ **do**
8: $crnt\_sys \leftarrow sys\_end + i$
9: $crnt\_app \leftarrow app\_end$
10: $ID\_sys\_delta \leftarrow getEventFromRECT($
$crnt\_sys, crnt\_sys, app\_start\_all, crnt\_app)$
▷ get delta at sys time
11: merge ID_sys_delta into ID_total
12: call $expression()$ ▷ callback for sliding sys
13: $j \leftarrow 1$
14: **while** $j <= N$ **do**
15: $crnt\_app \leftarrow crnt\_app + 1$
16: $ID\_app\_delta \leftarrow getEventFromRECT($
$sys\_start\_all, crnt\_sys, crnt\_app, crnt\_app)$
▷ get delta at app time
17: merge ID_app_delta into ID_total
18: call $expression()$ ▷ callback for sliding app
19: $j \leftarrow j + 1$
20: **end while** ▷ end sliding app
21: $i \leftarrow i + 1$
22: **end while** ▷ end sliding sys
23: **end procedure**

Table 4.3: Execution Sequence for Sliding Both Times

Seq#	AlgLn#	Event	DeltaRows	ValidRows
0	1	initial windows sys: [100,104] app: [10,12]	+3	3
1	10	slide sys at 105 sys: [105,105) app:[10,12]	-3+4	4
2	16	slide app at 13 sys:[100,105] app:[13,13)	-4+5	5
3	16	slide app at 14 sys:[100,105] app:[14,14)	-5+6	6
4	10	slide sys at 106 sys: [106,106) app:[10,14]	-6+7	7
5	16	slide app at 15 sys:[100,106] app:[15,15)	$\emptyset$	7
6	16	slide app at 16 sys:[100,106] app:[16,16)	-7	$\emptyset$
7	10	slide sys at 107 sys: [107,107) app:[10,16]	+8	8
8	—	—	—	—

1) when system time slides, and line 16 frames the application time delta (which is a vertical rectangle similar to Query Type 2) when the application time window slides.

Table 4.3 shows an execution sequence for the example in Figure 4.7. Seq 0 constructs the initial windows which are boxed in the rectangle. Seq 1 slides system time with the application window unchanged. Seq 2 and 3 slide application time, with the system window unchanged. Before Seq 4, the ratio $1:2$ is satisfied and it is time to move the system time window forward. The difference between Seq 4 and Seq 1 is the delta length was increased by application time sliding in Seq 2 and 3. Similarly, Seq 5 calculates the delta when the application time window slides, but the system time window for Seq 5 advances by 1 due to Seq 4, compared with delta

length in Seq 2 and 3. The deltas are highlighted in Figure 4.7.

4.5 Optimization 1: Non-Retroactive BiSW

In the previous sections, we assume that the sliding window slides forward, and each time it slides one unit. However, the assumptions do not necessarily have to be held for all cases. For the sliding directions, it is sometimes useful to slide backward (e.g. decrease $StartWin$ and $EndWin$). In addition, increasing and decreasing on $StartWin$ and $EndWin$ do not have to be at the same pace (e.g. increase $StartWin$ by 1 unit, and increase $EndWin$ by 2 units when a window slides). Furthermore, the direction and pace of the system time window and the application time window can be independently different. The BiSW structure described in previous sections is able to support sliding either direction (forward, or backward) at arbitrary system and application times, by slightly changing the algorithms. This flexibility comes from the application time partitioned system timeline index organization in BiSW.

If the workload slides the system time window forward only, and if the system time window always ends at the latest system time, we can improve the BiSW to a more space-efficient structure. We name the new structure as *non-retroactive* BiSW, while we call the comprehensive one discussed before as *retroactive* BiSW.

The major improvement of the non-retroactive BiSW is to keep all visible IDs updated and to expire previously-arrived data at maintenance time when new data arrives, while at the same time keeping the tuples partitioned by application time. Therefore no history is recorded in the structure. Figure 4.8 shows the non-retroactive BiSW counterpart to Figure 4.3. When new data arrives at system time 105, the update operation takes visible IDs at latest previous version (at system time 102), applies (in-)validations (for example, ID 3 is removed from the cell due to invalidation), and updates the time stamp to the new one, which is 105 in this example.

The non-retroactive BiSW has several benefits: for sliding system queries, Algorithm 1 line 11 which locates a particular system time and gets delta rows, is able to directly ingest the buckets as input because the visible IDs were updated at the time of maintenance; for sliding application time queries, Algorithm 2 line 8 does not need to construct a vertical rectangle which includes multiple cells, but instead the new status has already been updated into the cells in non-retroactive BiSW. However, the maintenance would take longer time compared with retroactive BiSW, becuase non-retroactive BiSW is no longer appended only, and has to remove invalided IDs from its cells. As a whole, non-retroactive BiSW trades off maintenance and the flexibility of supported queries, for a better space utilization.

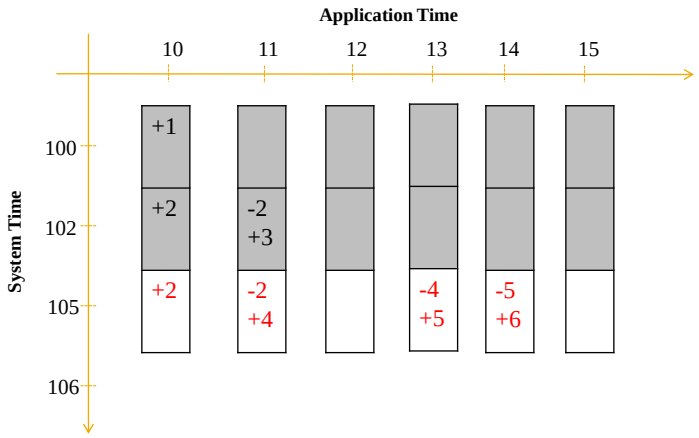


Figure 4.8: Non-Retroactive BiSW Updating (at system time 105)

4.6 Optimization 2: Checkpointing

In the retroactive BiSW when sliding the application time, the height of the delta rectangle starts from the overall system start time (recall the vertical rectangles in Figure 4.6), which implies a long aggregation especially if the system time has accumulated too long. A straightforward approach to save the aggregation time is to create *checkpoints* which keeps the all visible IDs at the checkpoint system time.

Figure 4.9 shows a checkpoint after system time 106. Therefore for a sliding application time query, the vertical delta does not need to start from system time 100. Instead, it can start from the nearest checkpoint time (106). Checkpointing saves the aggregation time for rectangles, especially when it has long system time edge. Checkpointing trades off space for better query performance.

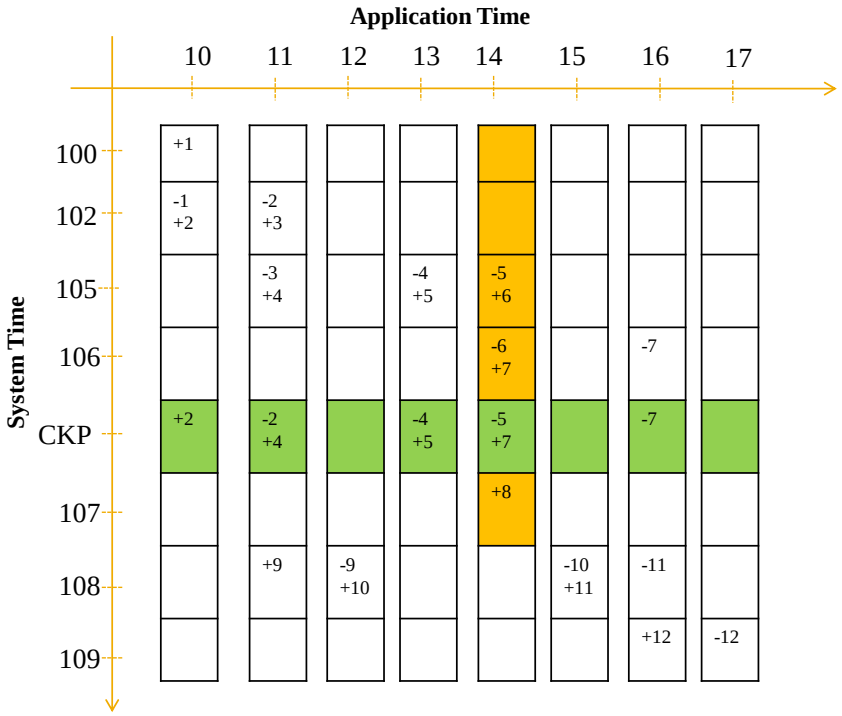


Figure 4.9: Checkpointing (at system time 106)

Chapter 5

Experiments and Results

In this section, we measure the performance of BiSW and compare it with several state-of-the-art alternatives. Section 5.1 gives details about the experimental environment and input; Section 5.2 describes the different comparisons and measurements. Experimental results are summarized in Section 5.3. Sections 5.4 to 5.7 explain the details.

5.1 Setup and Data Set

All experiments were carried out on a server with 256GB DDR3 1600MHz RAM and 2 Intel Xeon E5-2670 processors each with 16 cores, running a Linux operating system (kernel version 3.0.101). The prototypes were implemented entirely in C++. As the BiTL was designed to use intensively in memory, in order to compare fairly, all the contender data structures and raw table resided in memory.

The input data set was generated by the TPC-BiH benchmark [25]. The TPC-BiH benchmark takes the output from the standard TPC-H generator as system time version 1, and extends the non-temporal schemas to bitemporal schemas by adding four time attributes, as discussed in Section 2.1. Each tuple is associated with a system time interval as well as an application time interval. The TPC-BiH benchmark generates the application time with random distributions, and simulates the evolution of bitemporal tables by executing 9 update scenarios, where each transaction is associated with a system time. Each update scenario results in one transaction which generates a new version in the table. The application times are randomly distributed, and the system times except the initial system time version 1 have similar number of events per time, controlled by parameters in the TPC-BiH benchmark.

The experiments run on a bitemporalized TPC-H ORDERS table. The test data has the scaling factor SF_0 as 1 (a parameter in the TPC-BiH benchmark to control the size of table at system time 1), and has the scaling factor SF_H as 25 (a parameter in the TPC-BiH benchmark to control the size of updates). The raw size of the table is 10GB, which covers a range of 50 million system times and 10 million application times. More details of the data generation can be found in the TPC-BiH benchmark paper [25].

To simulate streaming scenarios, a data dispatcher was implemented to replay the update scenarios based on the final bitemporal table. The dispatcher sends a bunch of transactions (1000, in the following experiments) per time to the prototypes, and the prototypes update their index structures based on the new arriving data. After each update is finished, the following queries in Section 5.4 to Section 5.6 are issued to run based on the existing information in the prototypes. As data continuously arrives, and queries are issued to run again, so it is able to measure the scalability of different comparisons by the window size and table size.

As explained in Section 2.3, the maintenance cost is measured by the time to update the new arriving data into the index structures; the space footprints are measured as index size accumulation as time goes by; and the query performance is profiled by the time of constructing initial windows and the time of sliding windows.

5.2 Comparisons

BiSW (both retroactive and non-retroactive) is compared with the following four competitors. Note that the performance of the following comparison systems does not differ whether the system time window ends on largest system time or not. In order to conduct a more comprehensive comparison, we test the case that the system time window ends on largest system time when needed, for which case the non-retroactive BiSW can be compared as well.

- The Bitemporal Timeline Index (BiTL) [26] is designed to support general bitemporal queries such as aggregation and time travel. We implemented the sliding window framework on top of the BiTL.
- A B-tree representative. We adopt an open-sourced B+-tree [2] and have two independent B+-trees to index a bitemporal relation: one tree to index system time, and the other to index application time.
- A R-tree representative. As in Section 3, a 2-dimensional RR*-tree [9] is used to index bitemporal tuples: one dimensional indexes the system interval, and the other dimension

indexes the application interval. Our sliding window framework is implemented using the authors' original libraries [1].

- In addition, we also implement the sliding window queries using the raw table scan.

The query execution process of these competitors is described in detail at the Appendix. For the measurements, as explained in Section 2.3, we compare different implementations on maintenance overhead, query performance, and space footprint. All sliding window queries intentionally select row identifiers in order to query the index only and to avoid performance deviation of retrieving other columns.

5.3 Summary of Results

The results of experiments are as follows:

- *Memory footprint*. BiSW is around 50% smaller than the runner-up.
- *Maintenance*. BiTL is slightly better than BiSW, because of only maintenaning one single system timeline index,
- *Performance of getting initial window*. Before the window slides, the initial windows need to be constructed. This aspect measures how fast to get the initial window before it is able to slide. BiSW is 10x faster than the runner-up.
- *Performance of sliding system window*. This aspect measures how fast to slide the system time window. BiSW is 5x faster than the runner-up.
- *Performance of sliding application window*. This aspect measures how fast to slide the application time window. BiSW is approaching BiTL, which has the best performance in this case.
- *Performance of sliding both times*. This aspect measures how fast to slide both windows. We will show the ratio at $1:4$, and also the average performance at different ratios. BiSW is the best (with practical ratio from 1 to 100).

To bring all results together, our BiSW index outperforms all alternatives by all three measurements, with only two exceptions, where BiSW is only slightly worse than the best ones.

5.4 Experiment 1: Slide System Time

The first type of bitemporal sliding query is to slide the system time window with the application time window fixed. The query tested is the following:

```
select ROW_ID
from orders
    [SYS, START 0, RANGE sys_end/2, SLIDE 1]
    [APP, START 0, RANGE app_end/2, SLIDE 0]
    [RATIO 1:0]
```

The symbol sys_end denotes the largest system time in the bitemporal table, and it is growing as new arriving data is updated into the table. Since the TPC-BiH benchmark generates the application time in random distributions, hence the app_end is a fixed parameter during the whole history, representing the entire application time scope. For each sliding window query, the initial application time window is $[0, app\_end/2]$ and remains unchanged during system time sliding. The initial system time window for each system snapshot is $[0, sys\_end/2]$, and it slides 1 system time point per sliding. Every time when new data arrives, the index structures are updated, and the above query is issued to run, but with an increased sys_end value.

As described in Algorithm 1, the first step is to calculate the visible rows for the initial system and application time windows. Figure 5.1 shows the average execution time at different sys_end, and it measures the scalability as a function of system time window size. The x-axis represents the different sys_end, and the y-axis shows the execution time. The execution time of table scan grows linearly because the larger system time window size, the more rows it has to lookup. For the B+-tree, it first sequentially scans the application time tree from the beginning leaf node till the node representing $app\_end/2$ in order to get all application time visible ID set, and then it sequentially scans the system time tree until $sys\_end/2$ is reached to get all system time visible ID set. The final step is to intersect the two sets. For the RR*-tree, it just simply scans system time from 0 to $sys\_end/2$ and filters the application time interval on-the-fly. For the BiTL, at different $sys\_end/2$, it has to generate a new application timeline index in order to get the application visible IDs, which dominates the total execution time. Non-retroactive BiSW is better than retroactive one, because at each insertion time, it does not need to compute from system 0 to $sys\_end/2$ to get those valid rows.

Figure 5.2 zooms into the first 10 million system times, and it shows that both BiSWs outperform BiTL by an order of magnitude.

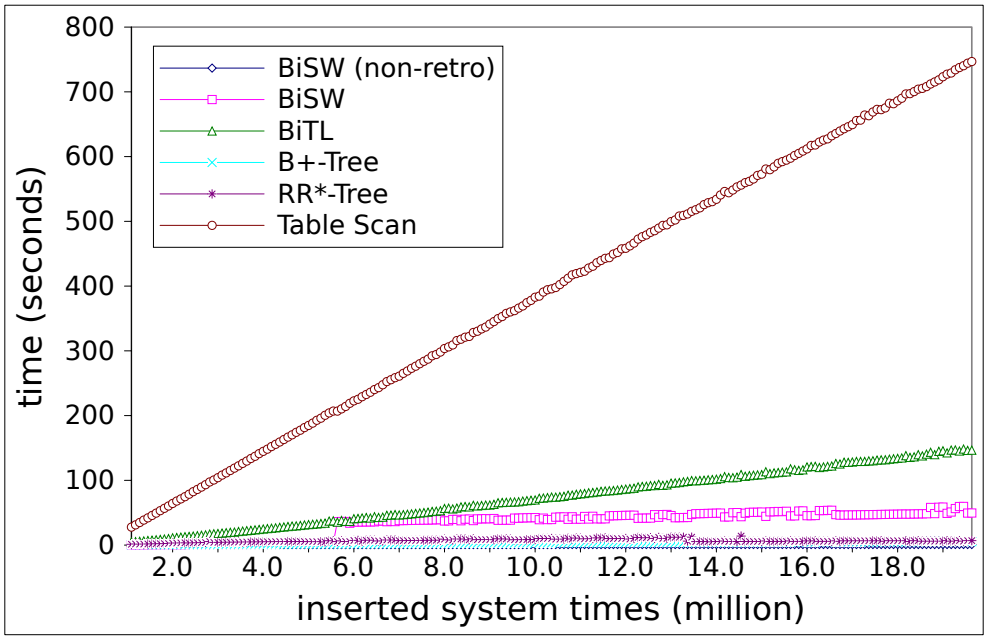


Figure 5.1: Time to Construct Initial Windows With Increasing sys_end (all)

After constructing the initial windows, the system time window begins to slide by computing a delta. All visible rows can be updated by merging the delta to the existing visible rows. Figure 5.3 shows the time for different comparisons to get a delta at different system times.

As explained earlier in Section 5.1, the number of events per transaction remains constant, therefore the incremental computation time remains constant for table scan. When system time slides, the B+-tree has to filter by its application tree in order to satisfy application time predicates, and because the events at each application time grows as new data comes in, therefore the B+-tree execution time grows. For the RR*-tree, the incremental computation is interpreted by matching those rectangles whose system time boundary coincides at a particular system time point. For BiTL when system time slides, BiTL has to create a delta application timeline, which degrades its performance compared with BiSW. Non-retroactive BiSW does not need to search the next system time from a full time timeline index, where the cells were updated before window slides.

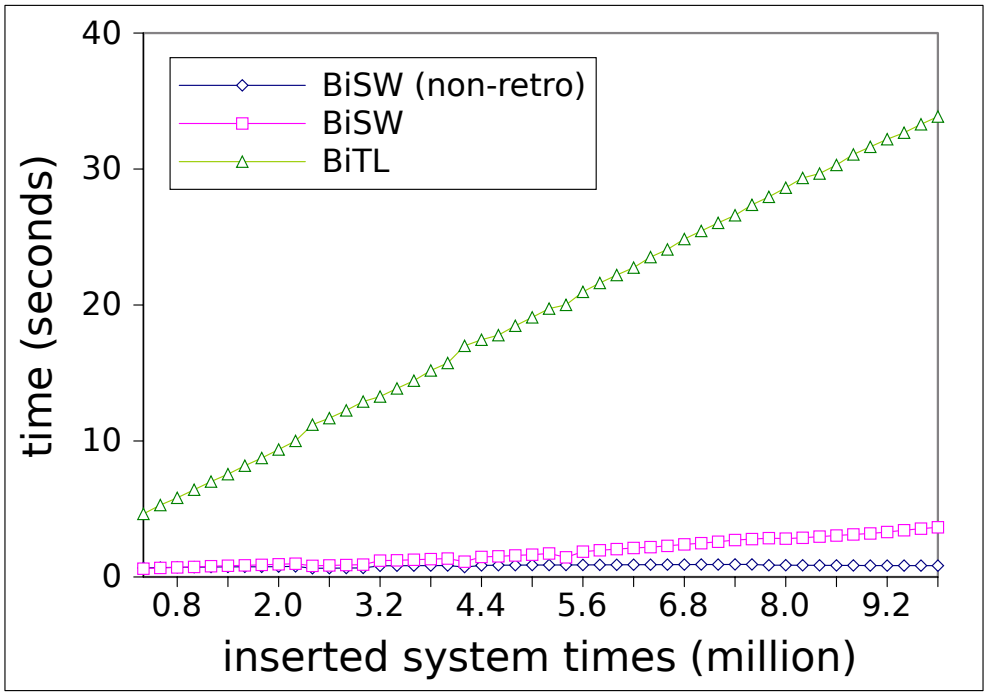


Figure 5.2: Time to Construct Initial Windows With Increasing sys_end (part)

5.5 Experiment 2: Slide Application Time

The second type of bitemporal sliding query is to slide the application time window with the system window fixed (at a particular system time snapshot). The query tested in this set is the following:

```
select ROW_ID
from orders
    [SYS, START 0, RANGE sys_end, SLIDE 0]
    [APP, START 0, RANGE app_end/2, SLIDE 1]
    [RATIO 0:1]
```

In this query, the initial system time window is $[0, sys\_end/2]$ and remains unchanged during the application time window sliding. The initial application time window is $[0, app\_end/2]$ and slides 1 application time per sliding.

The time to construct the initial windows with changing the app_end has similar linear property as in Experiment 1, so we do not show a separate graph for it. Every time when new data

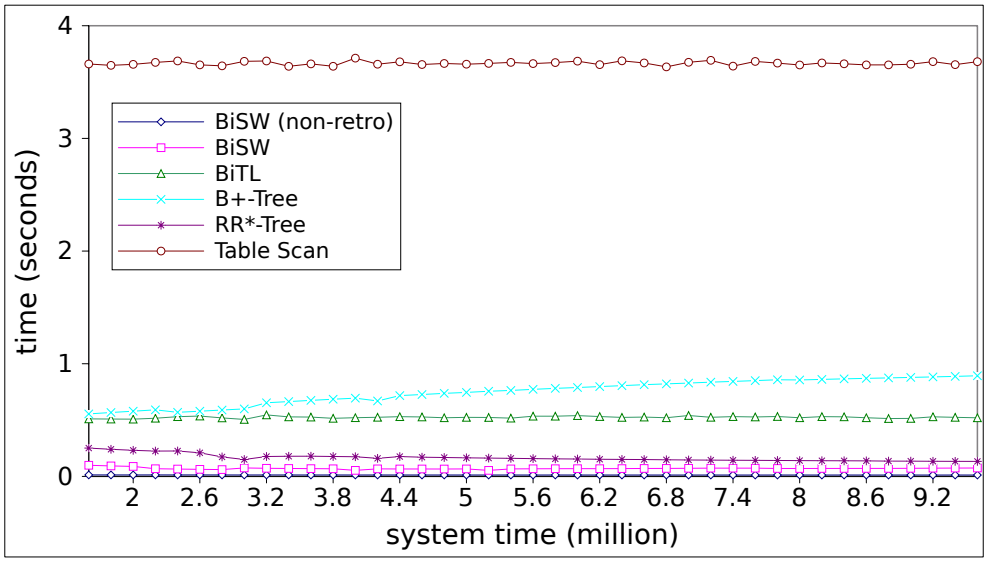


Figure 5.3: Time to Compute A Delta At Different System Times

arrives, the index structures are updated, and the above query is issued to run, but with an increased sys_end value. From the view at each application time, a larger sys_end indicates that on average more events occur at that particular application time, and that the delta size gets larger. Figure 5.4 shows the time to slide the application time window at different system time snapshots. The x-axis represents the inserted system snapshots at different system times, and the y-axis shows the total time to slide application time windows from the initial application time window to the last application time window instance ($[1 + app\_end/2, app\_end]$).

The execution time of retroactive BiSW grows as sys_end increases, because a larger sys_end means each partitioned system tineline index gets longer, which results in longer scan time. When the application time window slides, BiSW has to compute a delta from system time 0 to sys_end. The application timeline index in BiTL has the same valid rows as the corresponding non-retroactive BiSW, but it gains a slightly better performance compared with the non-retroactive BiSW, since the application timeline index in BiTL is not partitioned and has better cache coherence. In this experiment, we also create checkpoints at each 1 million sys_end for the retroactive BiSW, which virtually shortens the vertical delta by starting from last check-pointed system time instead of from the very beginning. It can be seen from Figure 5.4 that the execution time of retroactive BiSW drops at the check-pointed system times, and then increases as system time goes until another checkpoint is created.

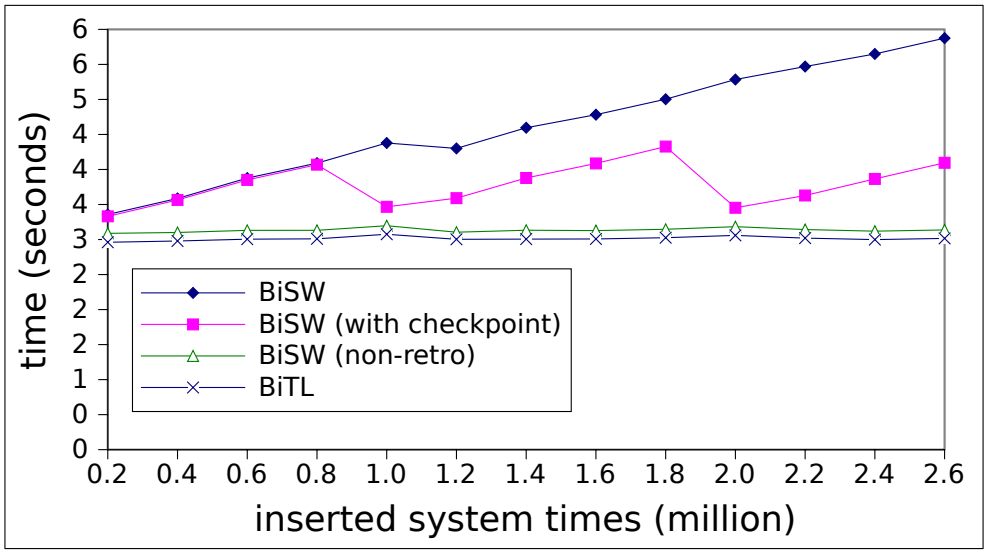


Figure 5.4: Time to Slide the Application Time Window At Different Snapshots

5.6 Experiment 3: Slide Both Times

The third type of bitemporal sliding query is to slide both the application time window and the system time window within one query. The query tested in this set is represented as:

```
select ROW_ID
from orders
    [SYS, START 0, RANGE sys_end/2, SLIDE 1]
    [APP, START 0, RANGE app_end/2, SLIDE 1]
    [RATIO 1:4]
```

The initial system time window is $[0, sys\_end/2]$, and the initial application time window is $[0, app\_end/2]$. The ratio of $1 : 4$ defines a sliding loop pattern: in each loop the system time window first slides once, and then the application time window slides 4 times.

Recalling the query evaluation algorithms in Chapter 4, the evaluation process of sliding window queries on both times is a combination of sliding the system time window and sliding the application time window, and the performance and algorithm for each sliding step do not differ from Section 5.4 and Section 5.5, hence we do not repeat the similar figures. Figure 5.5 shows an execution of a specific Experiment 3 query, where it ran on the largest sys_end (final bitemporal table after constructing the initial windows. In Figure 5.5, the query began to slide

both windows as specified by the *ratio* keyword: sliding system time window operations happen at each 5*N step. As explained in Experiment 1, when the system time slides, it is anticipated that BiTL consumes more time than BiSW because the application timeline index materialization dominates BiTL execution time. Figure 5.5 shows peaks when sliding system time windows occurred, and both the BiSWs outperform the other comparisons, which is consistent with the findings in Experiment 1. When application time slides, (non-retroactive) BiSW is approaching BiTL as explained in Experiment 2.

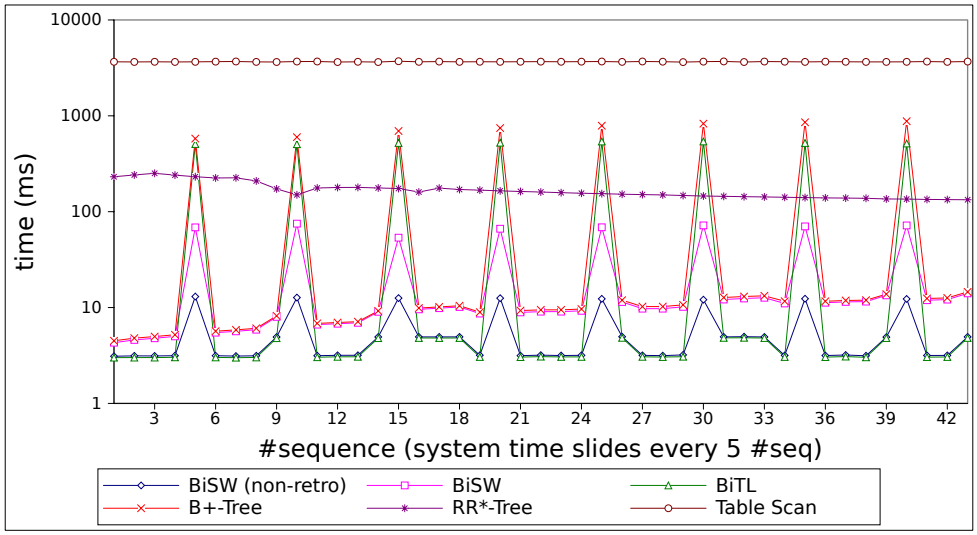


Figure 5.5: Bitemporal Sliding Sequence For The Final Bitemporal Table

Figure 5.6 shows the average time per sliding at different ratios. At the extreme rare case where there is no system time window sliding, BiTL is theoretically better than BiSW because BiTL is not partitioned. But in a practical mixed workload including sliding both times, the average cost of BiSW is the best.

5.7 Experiment 4: Maintenance Overhead

Figure 5.7 shows the bulk loading time as new data arrives. For the BiSW, the new data is partitioned into the corresponding timeline indices by its application time, so it takes longer time than the non-partitioned BiTL. But the difference between BiTL and BiSW is trivial compared with the apparent big gap to construct the initial windows, shown in Figure 5.2. For the non-retroactive BiSW, because it keeps no history and always updates to latest results, it takes longer

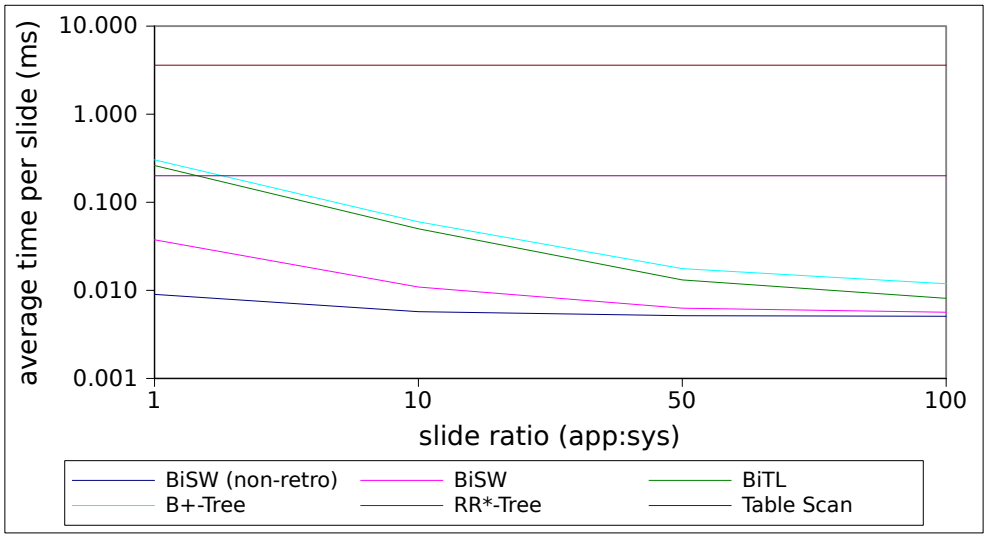


Figure 5.6: Average Time of Sliding Both At Different Ratios

time to insert and delete from the buckets. The B+-trees manages to insert validations and invalidation to the system time tree and the application time tree independently. The RR*-Tree inserts rectangles at the leaf nodes, and the minimum bound rectangles (directories) are managed by the RR*-tree itself. As shown in Figure 5.7, the BiSW maintenance time approaches the BiTL.

5.8 Experiment 5: Space Footprint

Figure 5.8 shows the size of different structures as new data comes in. It is obvious that the total size of different structures grows linearly as table size grows. BiTL and the B+-tree materialize the same row information twice: in the system index structure, and the other in the application index structure; therefore both of them have larger size than BiSW, which stores the same row only once. The RR*-tree is worse than BiSW because in addition to storing the rectangles of data, it also needs to store directories (the minimum bounding rectangle) as well.

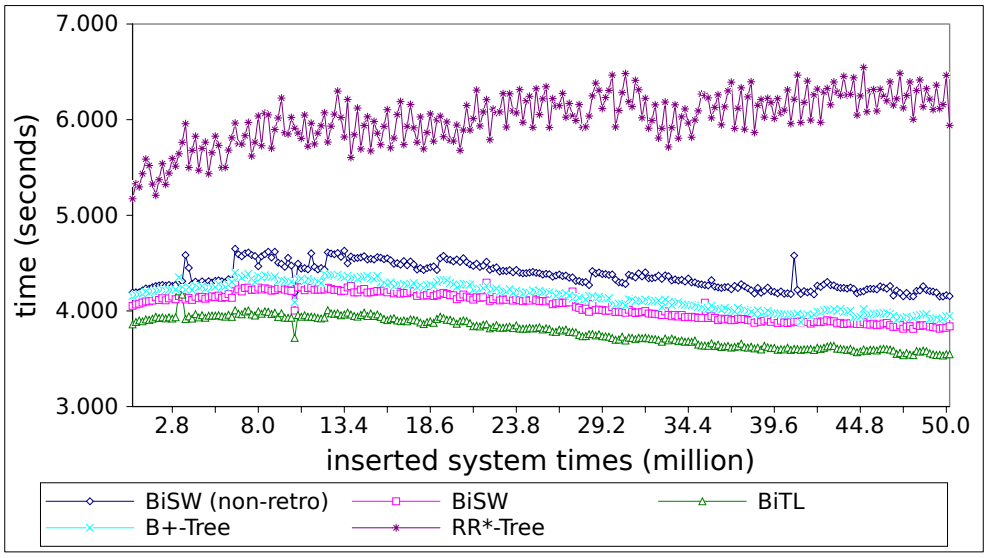


Figure 5.7: Maintenance Time For Arriving Data At Different System Times

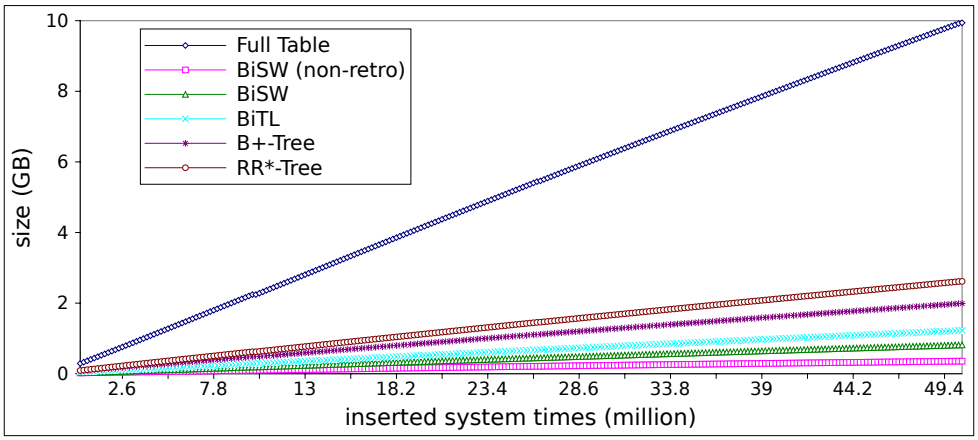


Figure 5.8: Structure Size As Data Arrives

Chapter 6

Conclusion

This work bridges the gap between two domains - the bitemporal data model and the sliding window model, and proposes an index structure called BiSW to seamlessly support bitemporal sliding window operations, which are very practical in the real world but on which not too many works have ever been conducted. Existing work on temporal and bitemporal indexing only focuses on using general indexes such as trees to index the temporal dimension, without considering the properties (e.g. incremental computation) of sliding windows, and also lacks efficient algorithms on how to answer sliding window queries. On the other side, despite a large amount of work that has been done on sliding windows, most of them assume the simpler system dimension only. Some of them indeed discuss application time properties, such as out-of-ordering; however these application time properties are not fully studied. There are few related works directly addressing the core problem: how to do sliding window operations on a bitemporal relation.

Therefore, this work proposes an index data structure and a series of algorithms to support the three categories of sliding window queries, both effectively and efficiently. Experiments show that BiSW outperforms its tree based competitors and state-of-the-art Bitemporal Timeline Index in most of the cases, usually by several times.

6.1 Future Work

Considering the flexibility and interactions of two time intervals, there is space to contribute to a more comprehensive understanding of bitemporal queries. A few directions are highlighted below:

- Benchmarking. Currently, there is only one bitemporal data benchmark (TPC-BiH [25]) which is able to generate a TPC-H table with bitemporal attributes. However, TPC-BiH considers general types of bitemporal workload such as time travel (restoring the table to a previous version), but it does not benchmark sliding window queries. In this paper, we classified three categories of bitemporal sliding window queries, and provides one classification of queries to a bitemporal sliding window benchmark.

- Sliding Query Language Support. The SQL:2011 Standard [31] includes the syntax to differentiate system time and application time; however there is no standard definition of (bi-)temporal sliding window query syntax. This paper gives an example of such query syntax, and it is desirable to come to a standard expression.

APPENDICES

Appendix A

Evaluation for Baselines

In this section, we describe how to evaluate bitemporal sliding window queries using the B-tree, R-tree, BiTL and raw table scan.

At the high level, all alternatives share the same workflow with the BiSW, e.g. a start condition has to be answered before incremental computation for bitemporal sliding window queries. They differ at the physical layout, which determines the query plan, maintenance time and space size.

A.1 B-Tree

In order to index both the system time and application time, two independent B-trees can be created to index time points, one for each time. In the leaf node, it stores the time point as key, and organizes RowID as a value chain (multiple rows can cause validation/invalidation at the same time point). For validations, the RowID is stored as a positive value, and for invalidations the opposite RowIDs (negative values) are recorded to distinguish from validations. B-tree organizes its keys in order (in this case to keep time in an ascending order) so that a lookup has a deterministic path from root to the target leaf node. Each leaf node is linked with neighbours to facilitate sequential leaf scan.

Figure A.1 shows the B-tree leaves to index system time corresponding to the table in Figure 3.1.

At system time sys_t a bunch of transactions comes in (recall that incoming data follows system time ordering). For the B-tree indexing system time, all incoming tuples go to the leaf

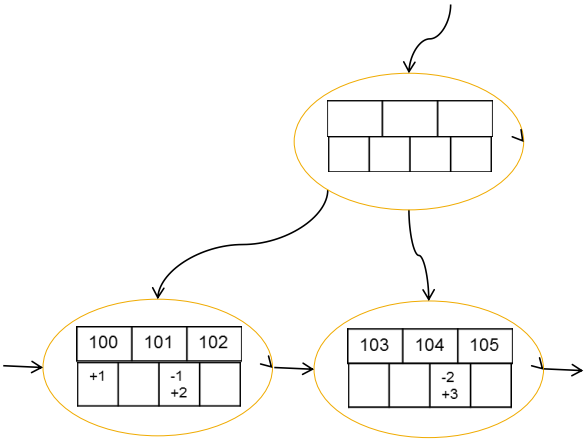


Figure A.1: The corresponding system B-tree (with Figure 3.1)

node containing sys_t, once that leaf has been located. Meanwhile the tree indexing application time also has to be updated, but maintenance on the application tree incurs more lookups: tuples may contain arbitrary application time intervals, resulting in an insertion of a positive RowID at its StartApp leaf, and another insertion of the negative RowID at its EndApp leaf, if EndApp is not ∞.

For sliding window queries, the start condition always has to be built from the overall start system and application time, which turns to be sequential scans on both trees from the left-most nodes until the right boundary of the initial window at each time dimension is reached. At this stage, the two sets contain all visible RowIDs from each perspective. Then the row identifier sets from both trees are intersected to get those row identifiers visible at both time dimensions, where the start condition completes.

Next for sliding one time (either system or application) dimension queries, the RowID set for the fixed interval time remains unchanged, and is used to filter out the other dimension. Taking the sliding system time queries for example, when the system time advances, the key for next system time in the B-tree leaf node is adjacent to the previous old system time, and locating the new key for next system time consumes constant time, simply by moving the pointer to the next key position. The associated value chain at the new system time key contains all RowIDs representing events at the new system time. The last step is to intersect with the application time RowID set, to filter out those RowIDs which are visible for the application time interval. A similar process applies to sliding application time queries. For sliding both times, none of the two RowID set is fixed. However, at each step, only one window changes with the other one fixed, which is used as the filter.

A.2 R-Tree

A 2-dimension Cartesian space R-tree can be used to index both temporal intervals at the same time. A rectangle which is bounded by its system and application time interval represents a row in the bitemporal relation. Each rectangle is associated with its RowID. For those tuples which have open-end intervals (e.g. End* is infinity), we reset the end boundary to the largest value where its time can reach. In addition, the R-tree also maintains the minimum bounding rectangles at higher levels in order to help searches. Figure A.3 shows an R-tree example with three leaf rectangle for the bitemoral table example in Figure A.2.

ID	Name	City	Balance	StartApp	EndApp	StartSys	EndSys
3	John	Largevill	40	11	12	102	105
4	John	Largevill	30	11	13	103	104
5	John	Costtown	100	13	14	105	107

Figure A.2: An Example of Bitemporal Table

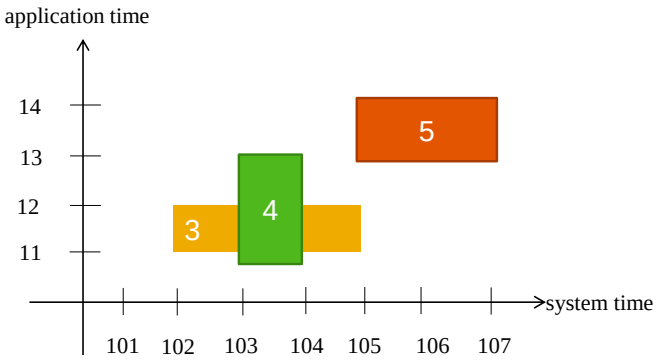


Figure A.3: An Example of R-tree (with Figure A.2)

For maintenance when new data arrives, a rectangle is inserted for every new validated tuple, and invalidating a row is implemented by setting the existing rectangle's system time right bound to be the time of deletion.

To answer the start condition on system time $[0, sys\_end]$ and on application time $[0, app\_end]$, a rectangle is created taking the two intervals from start condition. Those rectangles that cover the upper-right corner of the start condition (coordinate (sys_end, app_end)) imply their visibility: from the start condition's point of view, visible tuples should start their visibility before end boundaries of start condition, and continue to be valid until end boundaries are reached.

Incremental time slicing computation is quite straightforward: for example, a line which is vertical with system time axis at system time point t will intersect those tuples which are visible at t. In particular, intersecting with a rectangle on its boundary implies that representing row become valid/invalid at t, depending on whether intersecting the start or end boundary.

A.3 Bitemporal Timeline Index

As discussed in Section 3.3, the BiTL adopts lazy materialization for the application timeline index, and maintains the system timeline index only over system time. Initially there is no timeline structure to index application time until on-demand, and even if an application timeline index exists, it is not updated with the pace of system timeline.

So to answer the start condition, the system timeline index is scanned from the beginning if there is no application timeline index, or from the system time when the application timeline was created, until the start condition's system end point is reached. As system timeline scanning goes, a new application timeline index is generated, and merged if needed. The original paper [26] has more details on how to create a application timeline index.

After the application timeline index at the start condition's system end time is generated, the start condition can be answered using the application timeline index only, because the application timeline has already filtered out those invisible rows from system time's perspective (recall the conceptual view). In this case, a sequential scan from the beginning of the application timeline index to initial window's application end point captures all visible rows for the initial window.

For sliding system time queries, whenever the system time window moves, the application timeline index becomes out of date. In this case, an application timeline delta is created corresponding to those tuples at new system time point, and the delta is merged with the old application timeline index to upgrade to a new application time with new system time. Then the fixed application timeline can be used to scan the application timeline index.

For sliding application time queries, once the target application timeline is created, the BiTL is able to slide on application time efficiently because the application timeline index is implmented as an array. For queries sliding both times, the query plan combines the previous two processes: when the system time window slides, a delta application timeline is created and merged, and later the updated application timeline index is used to do the sliding on the application time.

A.4 Raw Table Scan

Table scan is naive and quite straightforward: there is little maintenance overhead, and it requires no additional space for auxiliary structures. However, query performance is significantly penalized, especially for queries involving application time, where a full table scan has to be performed.

Answering the start condition requires a full data scan, during which the temporal attributes are checked on the fly to see if they match the ranges. Sliding system time queries are able to take the advantages of transaction ordering, which avoids full data scan and restricts to scanning only a fraction of the table. However, because application time does not follow any ordering in a bitemporal table, therefore the incrementally sliding application time slice query is degraded to a full table scan for each time slice. This is tremendously expensive as the size of raw data can be very large, and growing with system time.

References

[1] RR*-Tree Libraries. http://www.mathematik.uni-marburg.de/~seeger/code/rrstar/.

[2] STX B+ Tree C++ Template Classes. http://panthema.net/2007/stx-btree/.

[3] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. Temporal Query Processing in Teradata. In *EDBT*, pages 573–578, 2013.

[4] Chuan-Heng Ang and Kok-Phuang Tan. The Interval B-Tree. *Inf. Process. Lett.*, 53(2):85–89, 1995.

[5] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream processing. In *CIDR*, pages 363–374, 2007.

[6] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.

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

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[9] Norbert Beckmann and Bernhard Seeger. A Revised R*-Tree in Comparison with Related Index Structures. In *SIGMOD Conference*, pages 799–812, 2009.

[10] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. In *VLDB*, pages 345–356, 1998.

[11] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. Light-Weight Indexing of General Bitemporal Data. In *SSDBM*, pages 125–138, 2000.

[12] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

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

[14] Ramez Elmasri, Gene T. J. Wuu, and Yeong-Joon Kim. The Time Index: An Access Structure for Temporal Data. In *VLDB*, 1990.

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

[16] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. DEMON: Mining and Monitoring Evolving Data. In *ICDE*, pages 439–448, 2000.

[17] Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[18] Lukasz Golab, Piyush Prahladka, and M. Tamer Özsu. Indexing Time-Evolving Data With Variable Lifetimes. In *SSDBM*, pages 265–274, 2006.

[19] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, pages 47–57, 1984.

[20] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[21] Christian S. Jensen, Leo Mark, Nick Roussopoulos, and Timos K. Sellis. Using Differential Techniques to Efficiently Support Transaction Time. *VLDB J.*, 2(1):75–111, 1993.

[22] Christian S. Jensen and Richard T. Snodgrass. Temporal Specialization and Generalization. *IEEE Trans. Knowl. Data Eng.*, 6(6):954–974, 1994.

[23] Christian S. Jensen and Richard T. Snodgrass. Temporal Data Models. In *Encyclopedia of Database Systems*. 2009.

[24] Martin Kaufmann, Peter M. Fischer, Norman May, and Donald Kossmann. Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past? In *EDBT*, pages 738–749, 2014.

[25] Martin Kaufmann, Peter M. Fischer, Norman May, Andreas Tonder, and Donald Kossmann. TPC-BiH: A Benchmark for Bi-Temporal Databases. In *TPCTC*, 2013.

[26] Martin Kaufmann, Chang Ge, Anil K. Goel, Norman May, Peter M. Fischer, and Donald Kossmann. Bitemporal Timeline Index: A Data Structure to Support Queries on Bitemporal Data in SAP HANA. In *VLDB under review*, 2014.

[27] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline Index:A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD Conference*, pages 1173–1184, 2013.

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

[29] Jürgen Krämer and Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *COMAD*, pages 70–82, 2005.

[30] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous Analytics over Discontinuous Streams. In *SIGMOD Conference*, pages 1081–1092, 2010.

[31] Krishna G. Kulkarni and Jan-Eike Michels. Temporal Features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[32] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Trans. Knowl. Data Eng.*, 10(1):1–20, 1998.

[33] Mario A. Nascimento and Margaret H. Dunham. Indexing Valid Time Databases via B+-Trees. *IEEE Trans. Knowl. Data Eng.*, 11(6):929–947, 1999.

[34] Ravi Rajamani. Oracle Total Recall / Flashback Data Archive. Technical report, Oracle, 2007.

[35] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *ICDE*, 2006.

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

[37] Albrecht Schmidt, Christian S. Jensen, and Simonas Saltenis. Expiration Times for Data Management. In *ICDE*, page 36, 2006.

[38] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, pages 507–518, 1987.

[39] Han Shen, Beng Chin Ooi, and Hongjun Lu. The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. In *ICDE*, pages 274–281, 1994.

[40] Narayanan Shivakumar and Hector Garcia-Molina. Wave-Indices: Indexing Evolving Databases. In *SIGMOD Conference*, pages 381–392, 1997.

[41] Richard Thomas Snodgrass et al. TSQL2 Language Specification. *SIGMOD Record*, 23(1), 1994.

[42] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *PODS*, pages 263–274, 2004.

[43] Yufei Tao and Dimitris Papadias. Efficient Historical R-Trees. In *SSDBM*, pages 223–232, 2001.

[44] Yufei Tao and Dimitris Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queri. In *VLDB*, pages 431–440, 2001.

[45] David Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *PODS*, pages 58–67, 1996.

[46] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.

[47] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.