Temporal Aggregation with Range Predicates

Donghui Zhang* Alexander Markowetz† Vassilis Tsotras‡
Dimitrios Gunopulos§ Bernhard Seeger¶

Notes to the editors and reviewers: A preliminary version of our result appeared in [ZMT+01]. From the conference version, this paper has made the following extensions:

- Provided lemma and theorem proofs.
- Addressed the functional range-temporal aggregation query, with both constant and general value functions.
- Presented a better reduction technique for the range-temporal aggregation query.
- Added intuitive examples.
- Extended related work.

*College of Computer & Information Science, Northeastern University, Boston, MA 02115. donghui@ccs.neu.edu
†Department of Computer Science, University of Science and Technology, Kowloon, Hong Kong. alexmar@cs.ust.hk
‡Computer Science Department, University of California, Riverside, CA 92521. tsotras@cs.ucr.edu. This work was partially supported by NSF IIS-9907477 and the Department of Defense.
§Computer Science Department, University of California, Riverside, CA 92521. dg@cs.ucr.edu. This work was partially supported by NSF CAREER Award 9984729, NSF IIS-9907477, the DoD and a gift from AT&T.
¶Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany. seeger@Mathematik.Uni-Marburg.de

Abstract

A temporal aggregation query is an important but costly operation for applications that maintain time-evolving data (data warehouses, temporal databases, etc.). Due to the large volume of such data, performance improvements for temporal aggregation queries are critical. Previous approaches have aggregate predicates that involve only the time dimension. In this paper we examine techniques to compute temporal aggregates that include key-range predicates as well (*range-temporal aggregates*). In particular we concentrate on the SUM aggregate, while COUNT is a special case. This problem is novel; to handle arbitrary key ranges, previous methods would need to keep a separate index for every possible key range. We propose an approach based on a new index structure called the *Multiversion SB-Tree*, which incorporates features from both the SB-Tree and the Multiversion B+-tree, to handle arbitrary key-range temporal aggregation queries. We analyze the performance of our approach and present experimental results that show its efficiency. Furthermore, we address a novel and practical variation called *functional* range-temporal aggregation. Here, the value of any record can be a general function over time. The meaning of aggregation is altered such that the contribution of a record to the aggregation result is proportional to the size of the intersection between the record's time interval and the query time interval. Both analytical and experimental results show the efficiency of our result.

Keywords: *temporal aggregates, indexing, range predicates, functional aggregation*

1 Introduction

With the rapid increase of the volume of historical data in data warehouses, temporal aggregation have become predominant operators for data analysis. Computing temporal aggregates is a significantly more intricate problem than traditional aggregation without the time dimension. This is because each database tuple is accompanied by a time interval during which its attribute values are valid. Consequently, the value of a tuple attribute affects the aggregate computation for all those instants included in the tuple's time interval.

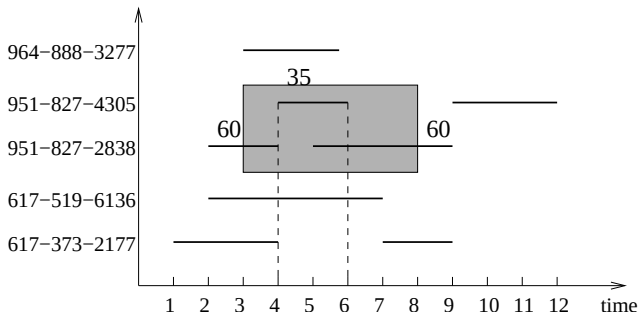


Figure 1: An example range-temporal aggregation query.

Consider the example of Figure 1. Here each horizontal line segment illustrates a phone call record. Such a record has a duration and a price. For instance, the record with unit price 35 cents started at time 4 and ended at time 6. One may be interested in knowing the total number of calls from the 951 area (Riverside, MA) during the time interval [3,8]. This query, illustrated by the shadowed query range, is an example of the plain range-temporal aggregation. Specifically, here the aggregation function is COUNT, and the query result is 3 since there are three records intersecting the query range. Another widely-used aggregation function is SUM. In this example the total price of the three call records is 35+60+60=155. Notice that the COUNT query is a special case of the SUM query when the value of every record is 1.

Sometimes, it is more meaningful if the query also takes into consideration of the length of intersection, between a record and the query range. Consider the phone call example in Figure 1 again. Suppose the price associated with each record is the unit price (per minute). The total cost of the phone calls intersecting the shadowed region should instead be 35*2+60*1+60*3=310.

More formally, this paper addresses the following problems:

Definition 1 *Let S be a set of temporal records, each record having a key, a time interval, and a value. Given a query key range R and time interval I:*

- *The* **plain range-temporal aggregation query** *computes the total value of records is S whose keys are in R and whose time intervals intersect I.*
- *The* **functional range-temporal aggregation query** *computes the weighted total value of records in S whose keys are in R and whose time intervals intersect I, where the weight on each record is the length of intersection between the record and I.*

Although there exists a large volume of work on temporal aggregation, this paper has different goals and/or models from all of them. Yang and Widom [YW01, YW03] proposed the SB-tree to address the temporal aggregation without a key range predicate. Our problem is different because we allow an arbitrary key range. The concept of temporal aggregation with range predicates was first introduced by us in a preliminary version of the paper [ZMT+01]. Since then there has been several follow-on work. Tao *et al.* [TPF04] addressed an approximate version of the range-temporal aggregation, with the goal to reduce the index size. Our goal is different for we focus on computing an exact answer. Nowadays, the price of hard drives keeps decreasing, while the storage size keeps increasing. Therefore it may be valuable to design an exact solution with the cost of a larger storage size (which is within a manageable bound). Kang *et al.* [KCK04] proposed the ITA-tree to address the range-temporal aggregation. However, it does not provide any guarantee on the worst-case query performance. In fact, since a query may need to visit multiple child nodes of an index node, the worst-case query performance should be linear (to the number of records). We aim at a method with very fast query support. Finally, Govindarajan *et al.* [GAA03] proposed the CRB-tree to address the range-aggregation query with multiple dimensions. They achieved a seemingly impossible index size, with the assumption of a *bit-wise* model. That is, any integer v is represented by $\log_2 v$ bits, and multiple integers may be compressed into a single word. As a consequence, in this model the implemented structures may have better theoretical bound on space, with the cost of significant implementation overhead. In this paper we assume the typical *word-wise* model, where each integer occupies a whole memory word.

Range-temporal aggregation is closely related to the *selection query*, which finds the records intersecting the query range. A large number of temporal indices have been proposed to efficiently support the selection query, as extensively surveyed in [ST99]. A straightforward approach to solve the range-temporal aggregation query is to utilize such an index to find the records intersecting the query range, and then aggregate their values on the fly. A major drawback however, is that the aggregation computation time is proportional to the number of records being selected. If many records satisfy the selection condition, the query performance can be as poor as linear search through the warehouse. For many applications, e.g. interactive analysis and decision making, such performance is prohibitive.

In this paper, we propose a new index structure which has guaranteed logarithmic query performance. We provide techniques that reduces the plain range-temporal aggregation query into several sub-

queries. We then propose a new index structure called the *Multiversion SB-Tree* (MVSB-tree) to solve these sub-queries. The proposed structure incorporates features from both the *SB-Tree* [YW01] and the *Multiversion B+-tree (MVBT)* [BGO+96]. By using two MVSB-trees we can maintain and compute plain range-temporal aggregation queries very efficiently. In particular, computing an plain range-temporal aggregate takes $O(\log_b n)$ I/Os, where b is the capacity of a disk page and n is the number of tuples in the warehouse. Updating the MVSB-tree is done incrementally as tuples are updated (an update takes $O(\log_b K)$ I/Os, where K is the number of different keys inserted into the warehouse). The space is bounded by $O(\frac{n}{b} \log_b K)$.

It is then illustrated that the functional range-temporal aggregation query can be solved using the same structure (the MVSB-tree) with constant augmentation on each index entry. Therefore the functional version of the query can also be solved with the same worst-case guarantee on query performance, update performance, and index size. We also extend our solution to the case when a record has as value not a constant, but a general function.

We compare the performance of our approach against using a temporal index that computes the aggregate on the tuples retrieved by the corresponding selection query. Possible choices for this index is a traditional multidimensional index (like an R*-tree [BKSS90]) or a temporal index (like the MVBT [BGO+96] or the TSB-tree [LS89]). We use the MVBT due to its efficiency. It was proved in [BGO+96] that the index can optimally compute a special case of the selection query, when the query time interval is reduced to a single time instant. And efficient selection queries with arbitrary query intervals exist [BS96]. Experimental results show that our approach provides superior performance in computing range-temporal aggregation queries at the expense of a small space overhead.

In summary, the contributions of this paper are:

- We propose the plain and functional range-temporal aggregation queries.
- We present techniques that reduce the plain range-temporal aggregation query into sub-queries.
- We propose a new index called the MVSB-tree, which solves the sub-queries. The index has guaranteed worst-case bound on query performance, update performance, and index size. In particular, while the straightforward solution to the range temporal aggregation problem has linear query performance, our proposed structure has logarithmic query performance.
- We solve the functional range-temporal aggregation query by augmenting the MVSB-tree. The solution can be applied both to the case when the value associated with a record is a constant, and to the case when a general value function is involved.

The rest of the paper is organized as follows. Section 2 discusses background and previous work. Section 3 presents a technique to reduce the plain range temporal aggregation problem to simpler problems. These simpler problems (and thus the plain range-temporal aggregation) are solved by the MVSB-tree index introduced and analyzed in section 4. Section 5 solves the functional range-temporal aggregation problem. Section 6 presents results from our experimental comparisons. Finally, section 7 concludes the paper.

2 Background

We first describe previous research on temporal aggregation queries including the SB-tree [YW01]. We then discuss the temporal data model assumed in our work and provide a short description of partially persistent indexing and in particular, the MVBT [BGO+96].

2.1 Related work on temporal aggregates

We consider four criteria for measuring the efficiency of a method that supports temporal aggregates. (1) The method should maintain the aggregates incrementally as tuples are inserted/updated. (2) The cost of inserting a new tuple should be independent from the tuple key and from the length of the tuple's interval. (3) The method should be disk-based, and, (4) the method should support not only *instantaneous* but *cumulative* temporal aggregates as well [YW01, MLI00]. The result of an instantaneous temporal aggregate at a given time instant is computed from the tuples valid at that instant. The value of a cumulative temporal aggregate at instant t is computed from the tuples whose intervals intersect interval $[t-w, t]$, for any given *window offset* w.

[Tum92] presents a non-incremental two-step approach where each step requires a full database scan. First the intervals of the aggregate result tuples are found and then each database tuple updates the values of all result tuples that it affects. This approach computes a temporal aggregate in $O(mn)$ time, where m is the number of result tuples (at worst, m is $O(n)$; but in practice it is usually much less than n). Note that this two-step approach can be used to compute range-temporal aggregates, however the full database scans makes it inefficient. [KS95] uses the aggregation-tree, a main-memory tree (based on the segment tree [PS85]) to incrementally compute temporal aggregates. However the structure can become unbalanced which implies $O(n)$ worst-case time for computing a scalar temporal aggregate. [KS95] also presents a variant of the aggregation tree, the k-ordered tree, which is based on the k-orderness of the base table; the worst case behavior though remains $O(n)$. [GHR+99, YK97, GGM+04] introduce parallel extensions to the approach presented in [KS95]. [MLI00] presents an improvement by considering a balanced tree (based on red-black trees). However, this method is still main-memory resident.

Yang and Widom [YW01, YW03] proposed the SB-tree, an incremental and disk-based index structure, to address the temporal aggregation without a key range predicate. We introduced the concept of temporal aggregation with range predicates in a preliminary version of the paper [ZMT+01]. Recently Tao *et al.* [TPF04] addressed an approximate version of the range-temporal aggregation, with the goal to reduce the index size. Kang *et al.* [KCK04] proposed the ITA-tree to address the range-temporal aggregation. However, it does not provide any guarantee on the worst-case query performance. In fact, since a query may need to visit multiple child nodes of an index node, the worst-case query performance should be linear (to the number of records).

Also related is the work on multi-dimensional aggregations with range predicates. We addressed the aggregation problem with multiple dimensions, where the objects may or may not have extent, in [ZTG02]. Govindarajan *et al* proposed the Compressed Range B-tree (CRB-tree) [GAA03], which is an external version of the Compressed Range tree [Cha88]. The CRB-tree uses $O(n)$ disk blocks and answers two-dimensional range-COUNT queries in $O(\log_B n)$ I/Os. Given a set of N points in 2-dimension space P and a query rectangle R, the range-COUNT query is to compute the total number of points in R. The solution was extended to handle Range-SUM queries. The basic idea is to store the weights along with the secondary structure. Their solution achieves near linear space

cost and $O(log_B n)$ I/Os query cost. To achieve such good space and query cost, [GAA03] assumed the *bit-wise* model. That is, any integer v is represented by exactly $\log_2 v$ bits. This model results in index structures difficult to implement. Furthermore, dynamic updates may deteriorates the performance. Also related is [GAE00] which proposes the *dynamic data cube*. It addressed the problem of answering multidimensional datacube range-sum queries.

Recently, [BGJ06b] extended multi-dimensional aggregation to apply to data with valid-time intervals. A new aggregation operator was suggested. [BGJ06a] provided a general framework of temporal aggregation and discussed the abilities of five approaches to the design of temporal query languages with respect to temporal aggregation.

2.2 The SB-tree

Since our proposed index structure, the MVSB-tree, draws ideas from the SB-tree [YW01, YW03], let's review the SB-tree in more detail.

The SB-tree incorporates properties from both the segment tree [PS85] and the B-tree. The segment tree features ensure that the index can be updated efficiently when tuples with long intervals are inserted or deleted. The B-tree properties make the structure balanced and disk-based. Conceptually the SB-tree indexes the time domain of the aggregated tuples. Each interior tree node contains between $b/2$ and b records, each record representing one contiguous time interval. For each interval, a special value is also kept in the record that will be used to compute the aggregate over this interval. Intervals are kept in both interior and leaf nodes. Moreover, the overall interval associated with a node contains all intervals in the node's subtrees.

An advantage of [YW01] is that an instantaneous temporal aggregate is computed by recursively searching the SB-tree (starting from the root) and accumulating the aggregate value along the tree nodes visited. This results in fast aggregate computation time, namely, $\mathrm{O}(\log_b n)$. Note that a special "compaction" algorithm is also presented that merges leaf intervals with equal aggregate values. This can reduce the height of the tree and hence its aggregate computation to $\mathrm{O}(\log_b m)$.

The second advantage of the SB-tree is its fast update time, which is also logarithmic. The insertion of a new tuple with interval i and attribute value v is first directed into the root node. Each root record whose time interval is fully contained in i is updated by value v (the kind of update depends on the aggregate maintained by the SB-tree). Whenever interval i is partially contained by a root record, it is recursively inserted in the subtree under this root record. The SB-tree allows physically deleting tuples from the warehouse. Such a deletion is represented as an insertion of a new tuple with a negative attribute value v.

To support cumulative SUM, COUNT and AVG aggregates with arbitrary window offset w, two SB-trees are used, one maintaining the aggregates of records valid at any given time, while the other maintaining the aggregates of records valid strict before any given time. To compute the aggregation query, the approach first computes the aggregate value at the end of interval w. It then adds the aggregate value of all records with intervals strictly before the end of w and finally subtracts the aggregate value of all records with intervals strictly before the beginning of w. Finally, we note that a special extension of the SB-tree (the min/max SB-tree) can be used to support MIN and MAX aggregates, too.

2.3 Temporal Data Model

For simplicity, we assume that each tuple in the warehouse is stored as a record that contains a *key*, a *time interval* and an attribute whose *value* is to be aggregated. We follow the *First Temporal Normal Form (1TNF)* [SS88] which specifies that there are no two tuples with equal keys and intersecting intervals. Without loss of generality, assume the key space is $[0, maxkey]$ and the time space be $[0, maxtime]$.

When considering temporal data, it is important to distinguish the time model used by the temporal application. In the temporal database literature two time dimensions have been proposed, namely the valid-time and the transaction-time [Jen98]. The kind of updates supported on the temporal data depends on whether valid-time or transaction-time (or both) is supported [KTF98]. In a valid-time environment when a tuple is inserted in the database, its associated interval is fully known. Moreover, tuples can be added and deleted from the database in any order. After a tuple is deleted its record is physically removed from the database (and thus cannot be further queried). The SB-tree has been designed for the valid-time environment.

In contrast, a transaction-time environment assumes that tuple updates arrive in the database ordered by time. Hence, when a tuple is inserted at time t_i, its record's interval is initiated as $[t_i, now)$ where *now* is a variable representing the ever increasing current time (in practice, variable *now* is stored as *maxtime*). However, a tuple deletion is not physical but logical. For example, if the above tuple is deleted at time t_j its record's interval *end* is updated from *now* to t_j. That is, the record is still maintained in the database and can be queried. Since deletions are logical, in a transaction-time environment we cannot change the past. Equivalently, the transaction-time model maintains the history of a time-evolving database. The ability to change the past is useful in cases where errors are discovered in the recorded information.

In this paper we assume that the warehouse follows the transaction-time model. We feel that this is a practical scenario since in many applications changes arrive in their time order. Furthermore, in our view, the number of erroneous tuples in a data warehouse is much smaller than the correct ones and, if needed, any corrections can be kept separately. Moreover, few errors are usually not important when considering aggregate values over a large number of tuples. Assuming the transaction-time model has a major influence on the index used to support aggregate queries. Since updates arrive in order, the index does not have to order them.

2.4 Partially Persistent B-trees

A data structure is called *persistent* if an update creates a new version of the data structure while the previous version is still retained and can be accessed. If the old version is discarded, the structure is called *ephemeral*. *Partial persistence* implies that updates are applied only at the latest version of the data structure, creating a linear version order. Clearly, partial persistence fits nicely with the notion of transaction-time; version numbers can be replaced by the ordered sequence of time instants. As we will show, the MVSB-tree is a SB-tree made partially persistent. Our approach has been influenced by the MVBT [BGO$^+$96] which is a structure that makes a B+-tree partially persistent.

Conceptually, the MVBT is a graph that maintains the evolution of a B+-tree over time. It has many roots, each responsible for accessing the B+-tree as it was during a specific time interval. The MVBT partitions the key-time space into rectangles where each rectangle is associated with exactly

one data page. A tuple's record is stored in all the data pages whose key-time rectangle contains the tuple's key and intersects its interval. The page rectangles are created recursively. As records are inserted into a certain page of a MVBT, this page may overflow. Then, the page's alive records are copied to another page. The kind of copying is based on the number of alive records in the overflowed page. A *time split* simply copies all alive records into a new page. If many alive records exist, the time split is followed by a *key split* that distributes them into two new pages according to the median of their key attribute.

Data records are inserted in the MVBT in increasing time order. An important feature of the MVBT is that it guarantees a minimum key density for every page. In particular, for any time t in the page's rectangle, the page contains at least d records that are alive at t, where d is linear to the page capacity. If after a deletion, the key density of the page drops below the threshold d (*weak underflow*), the alive records in the page and a sibling page are copied into a new page. To avoid frequent merge/splits, the number of records in a new page must be between a lower bound and a higher bound (*strong condition*).

The MVBT optimally solves (in linear space) the range-snapshot query: "find all tuples with keys in range r that were alive at time t". If the query answer has size s, the MVBT finds this answer in $O(\lceil \log_b n + s/b \rceil)$ I/Os.

3 Problem Reduction for the Plain Range Temporal Aggregation

In the preliminary version of this paper [ZMT+01] we have proposed a reduction technique which reduces one plain range temporal aggregation query to six dominance-sum queries. Here in Section 3.1 we propose a new and better reduction technique which reduces a range temporal aggregation query to four dominance-sum queries. The old reduction technique is described in Section 3.2. To get an overall picture, Section 3.3 provides the indexing scheme, update algorithm and query algorithm, with the assumption that a dominance-sum index exists. Later in Section 4 we propose the MVSB-tree that supports efficiently the dominance-sum query.

3.1 The New Reduction Technique Needs Four Dominance-Sum Queries

This section first defines the dominance-sum query, and then proposes a reduction approach that reduces a plain range-temporal aggregation query to four dominance-sum queries.

Definition 2 *Let P be a set of point objects in the 2D time-key space. Each record has a time instant, a key, and a value. A query point $q = (t, k)$* **dominates** *all objects with time instants smaller than t and with keys less than k. The* **dominance-sum query** *computes the total value of objects in P dominated by q.*

As an example, in Figure 2 the query point q dominates two objects (in the shadowed region) with values 2 and 3, respectively. Therefore the dominance sum is 2+3=5.

Theorem 1 *The plain range-temporal aggregation query is reduced to four dominance-sum queries.*

Proof. Intuitively, a query time interval and a key range collectively form a query rectangle Q in the 2-dimensional time-key space. The plain range-temporal aggregation query asks for the total value of objects intersecting Q. The query rectangle Q has 4 corners. A range-temporal aggregation query

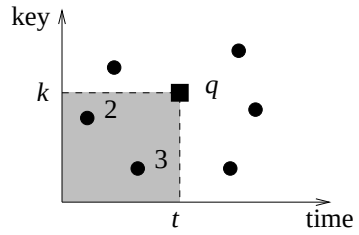


Figure 2: Illustration of the dominance-sum query.

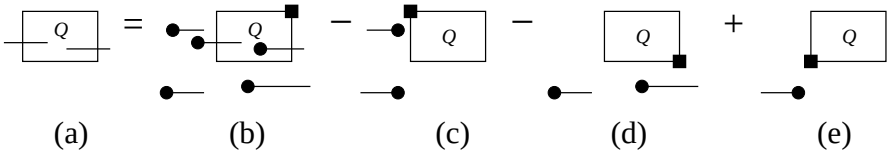


Figure 3: A plain range-temporal aggregation query is reduced to 4 dominance-sum queries.

is then reduced to 4 dominance-sum queries, one for each corner of the query box. This is illustrated in Figure 3. In particular, Figure 3(a) shows a query rectangle and two records intersecting with it. The range-temporal aggregation query asks to compute the total value of these two records.

To prove the theorem, we first note that in order for a record o to intersect the query rectangle Q, the left corner of o has to be dominated by the upper right corner of Q. Figure 3(b) shows the candidate records. Some candidates are false positives since they are either completely to the left, or completely under, Q. The false positives to the left of Q correspond to those whose right corners are dominated by the upper left corner of Q (Figure 3(c)). The false positives under Q correspond to those whose left corners are dominated by the lower right corner of Q (Figure 3(d)). After these false positives are subtracted from the query result, the objects whose right corners are dominated by the lower left corner of Q (Figure 3(e)) are subtracted twice. So their sum must be added again.

To sum up, if there exists an index that efficiently computes dominance sums, we can use two such indices to support the range-temporal aggregation in the following way. Maintain a dominance-sum index for the left corners of all records, and a separate dominance-sum index for the right corners of all records. A range temporal aggregation query is reduced to two dominance-sum queries on the left corners and two dominance-sum queries on the right corners. □

3.2 The Old Reduction Technique Needs Six Dominance-Sum Queries

Our old reduction technique [ZMT+01] reduces a plain range-temporal aggregation query into two LKST queries and four LKLT queries. This section first describes the reduction technique, then links the LKST and LKLT queries with the dominance-sum query.

Definition 3 *Let S be a set of temporal records, each record having a time interval, a key, and a value. Given a time instant t and a key k,*

- *The* **less-key, single-time (LKST) query** *computes the total value of records is S whose keys are less than k and whose time intervals contain t.*

- *The* **less-key, less-time (LKLT) query** *computes the total value of records in S whose keys are less than k and whose time intervals are strictly before time t.*

Theorem 2 *The plain range-temporal aggregation query is reduced to two LKST and four LKLT queries.*

Proof. Let the query key range be $R = [k_1, k_2)$ and the query time interval be $I = [t_1, t_2)$. If we only consider tuples with keys in R, the total value of tuples whose intervals intersect I is equal to the the total value of those tuples alive at t_2, plus the total value of those tuples alive strictly before t_2, and minus the total value of those tuples alive strictly before t_1. This can be described by the following equation:

$$SUM(R, [t_1, t_2]) \quad = \quad SUM(R, t_2) + SUM(R, end \leq t_2) - SUM(R, end \leq t_1)$$

We now consider all the tuples alive at t_2. $SUM(R, t_2)$ can be computed as the total value of the tuples whose keys are less than k_2 minus the SUM of the tuples of the records whose keys are less than k_1. Or,

$$\begin{aligned} SUM(R, t_2) &= SUM(key < k_2, t_2) - SUM(key < k_1, t_2) \\ &= LKST(k_2, t_2) - LKST(k_1, t_2) \end{aligned}$$

Similarly, we have:

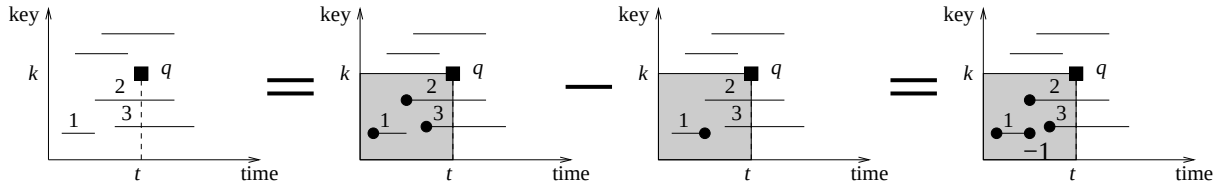
$$\begin{aligned} SUM(R, end \leq t_2) &= LKLT(k_2, t_2) - LKLT(k_1, t_2) \\ SUM(R, end \leq t_1) &= LKLT(k_2, t_1) - LKLT(k_1, t_1) \end{aligned}$$

Hence, we get:

$$\begin{aligned} PRTA([k_1, k_2), [t_1, t_2)) &= LKST(k_2, t_2) + LKLT(k_2, t_2) + LKLT(k_1, t_1) \\ &- LKST(k_1, t_2) - LKLT(k_1, t_2) - LKLT(k_2, t_1) \end{aligned} \quad (1)$$

Here $PRTA$ stands for plain range-temporal aggregation. Clearly, a range temporal aggregation query is reduced to two LKST queries and four LKLT queries. □

Interestingly, both the LKLT query and the LKST query can be mapped to the dominance-sum query. First, it is easy to see that the LKLT query is exactly the dominance-sum query, if we consider the right corners of all records.



(a) LKST result is 5 (b) DS on start points is 6 (c) DS on end points is 1 (d) unified DS is 5

Figure 4: The LKST query is also reduced to the dominance-sum query.

To see that the LKST query can also be reduced to the dominance-sum query, consider Figure 4. In Figure 4(a), the LKST result, corresponding to $q = (t, k)$, is 2+3=5. The reason is that there are two records with values 2 and 3 whose time intervals contain t and whose keys are less than k. Notice that the start points of these two records are dominated by q. But the dominance-sum of q on the start points of the records (Figure 4(b)) is more than what's needed. The difference

is the dominance-sum of q on the end points of the records (Figure 4(c)). By combining the two cases, we can keep the start points and end points of all records in one dominance-sum structure, where the value for each end point is negative to the original value. An LKST query result is the dominance-sum of these end points (Figure 4(d)).

3.3 The Overall Picture

Before describing the MVSB-tree in Section 4 which keeps a set of 2D point objects and efficiently supports the dominance-sum query, this section summarizes the two above two approaches in terms of indexing, insertion, and query schemes supporting the plain range temporal aggregation query, utilizing the MVSB-tree as basic building block.

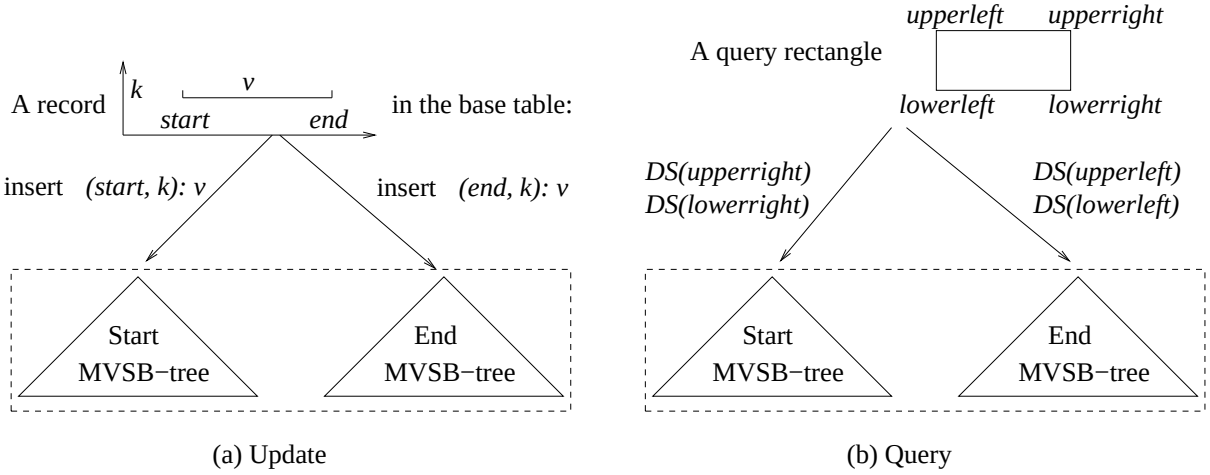


Figure 5: Illustrating of the *New* approach.

The *New* approach is illustrated in Figure 5. It uses two MVSB-trees, one corresponding to the start points of the original temporal records and the other corresponding to the end points of the original temporal records. For ease of presentation let's name the two MVSB-trees as the *Start MVSB-tree* and the *End MVSB-tree*. For one temporal record with key=k, time interval=[*start*, *end*), and value=v, two updates are needed, one in each MVSB-tree. The record's start point is inserted into the Start MVSB-tree as a point object, and its end point is inserted into the End MVSB-tree. To answer one plain range-temporal aggregation query, four dominance-sum (DS) queries are performed, two in each MVSB-tree. Let the four corners of the query rectangle be $lowerleft$, $lowerright$, $upperleft$, and $upperright$. As illustrated in Figure 3, the aggregation result is calculated as:

$$DS_{Start}(upperright) - DS_{End}(upperleft) - DS_{Start}(lowerright) + DS_{End}(lowerleft)$$

The *Old* approach is illustrated in Figure 6. It uses two MVSB-trees, one corresponding to the LKLT query and the other corresponding to the LKST query. Let's name the two MVSB-trees as the *LKLT MVSB-tree* and the *LKST MVSB-tree*. For one temporal record with key=k, time interval=[*start*, *end*), and value=v, three updates are needed. The start point is inserted into the LKLT MVSB-tree, while both the start point and the end point are inserted into the LKST index. In particular, the end point inserted to the LKST index has negative value. To answer one plain range-temporal aggregation query, six dominance-sum (DS) queries are performed, two in the LKLT MVSB-tree and

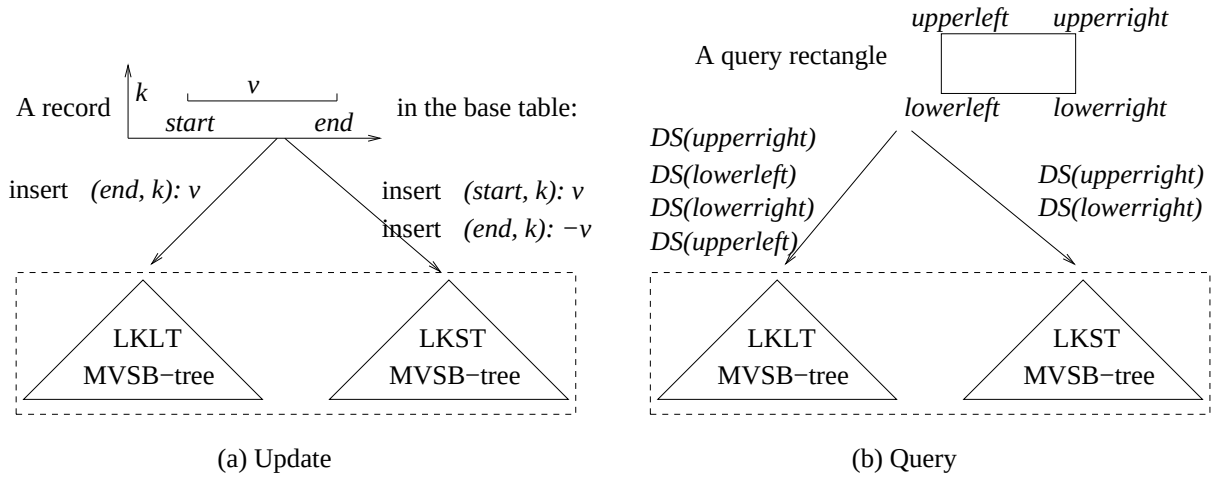


Figure 6: Illustrating of the *Old* approach.

four in the LKST MVSB-tree. As Equation 1 shows, the aggregation result is:

$$DS_{LKST}(upperright) + DS_{LKLT}(upperright) + DS_{LKLT}(lowerleft)$$
$$-DS_{LKST}(lowerright) - DS_{LKLT}(lowerright) - DS_{LKLT}(upperleft)$$

4 The Multiversion SB-tree

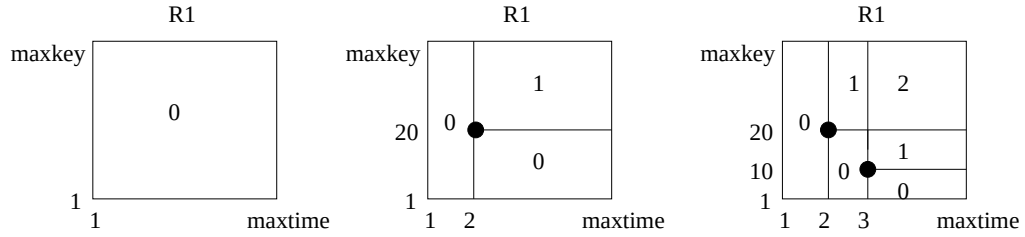
This section presents the Multi-version SB-tree (MVSB-tree), which is a disk-based, paginated, and dynamically updateable index structure that efficiently supports the dominance-sum query. The index supports, in logarithmic time, the update and query operations as described below.

- An update is to insert a 2D (time and key) point object (t, k) with value v into the index. According to the transactional time model, objects are inserted in non-decreasing time order.
- A query is to find the total value of objects, ever inserted into the index, which are dominated by some query point in the time-key space.

4.1 The Leaf Page of an MVSB-tree

Figure 7(a) illustrates the initial MVSB-tree with no object inserted. It has a a single page R_1 which is both a root page and a leaf page. Inside R_1, a single record is stored. The record has a rectangle which is the whole time-key space, and a value with is 0. This value is the dominance-sum query result for any query point that falls inside of the rectangle of the record. Since no object has been inserted yet, a dominance-sum query should indeed return 0.

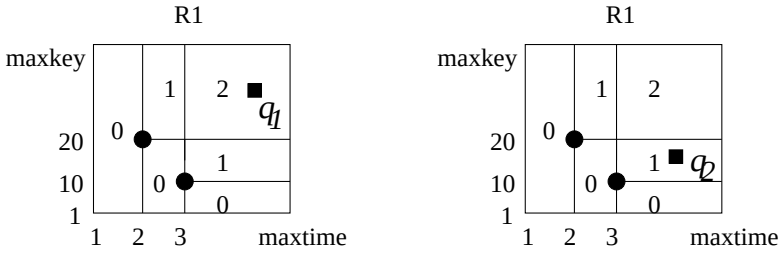
Suppose an object $\langle 2, 20\rangle : 1$ has been inserted (Figure 7(b)). It is a point object (2, 20) with value 1. The dominance-sum query result should be 1 if the query point is located to the upper right of (2, 20), and should be 0 otherwise. The MVSB-tree implements this by splitting the original record into



(a) the initial MVSBT (b) after inserting <2, 20>:1 (c) after inserting <3, 10>:1

Figure 7: The initial MVSB-tree and the layout after one and two insertions.

three. The rectangles of the three records do not overlap, and the union of them covers the whole space. The value of each record remains to be the dominance-sum query result for query points that fall inside the record's rectangle. To continue the example, Figure 7(c) shows the layout of the MVSB-tree after inserting $\langle 3, 10\rangle : 1$.



(a) q1=(5, 30) dominates 2 objects (b) q2=(5, 16) dominates 1 object

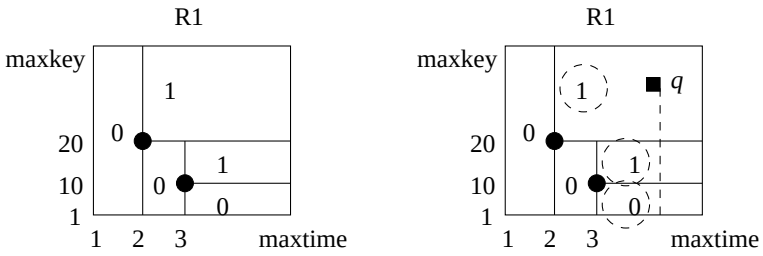
Figure 8: The value associated with a record in the MVSB-tree indicates the dominance-sum query result for a query point that falls into the record's rectangle.

To see that the dominance-sum query results are stored in the MVSB-tree, examine Figure 8(a). Here the query point q_1 dominates both objects inserted. So the dominance-sum query result is the total value of these two objects, which is 2. Indeed the record whose rectangle contains q_1 has value 2. Figure 8(b) shows another dominance-sum query, with query point q_2. It dominates one of the inserted objects. Indeed the record whose rectangle contains q_2 has value 1.

4.2 The Logical Splitting Optimization

In Figure 7(b), the right border of the rectangle (which corresponds to *maxtime*) has two segments: one with key range [1, 20) and the other with key range [20, *maxkey*). Each segment corresponds to a different record store in the page. A new insertion may split both records as shown in Figure 7(c). In general, one insertion may split many such records. This brings enormous space cost.

We hereby propose the **Logical Splitting Optimization (LSO)** which splits a single record, but is logically equivalent to splitting multiple records. The idea is that we only split the record whose key range contains the key of the new object. This split physically adds a value to only one record. The counterpart of Figure 7(c) with LSO is shown in Figure 9(a). In order to produce the correct dominance-sum query result, the query result is produced by aggregating the values of records below



(a) page layout with LSO (b) query result is 1+1+0=2

Figure 9: The counter part of Figure 7(c) with LSO. An insertion splits one record. A query gets accurate result by aggregating the values of records below the query point.

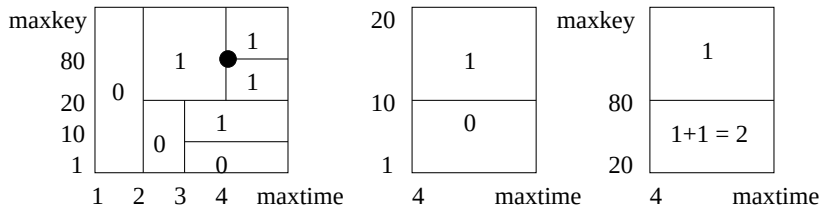
the query point. This is illustrated in Figure 9(b). The dominance-sum of the query point q is computed by aggregating the values 1, 1, and 0.

4.3 The Complete Structure

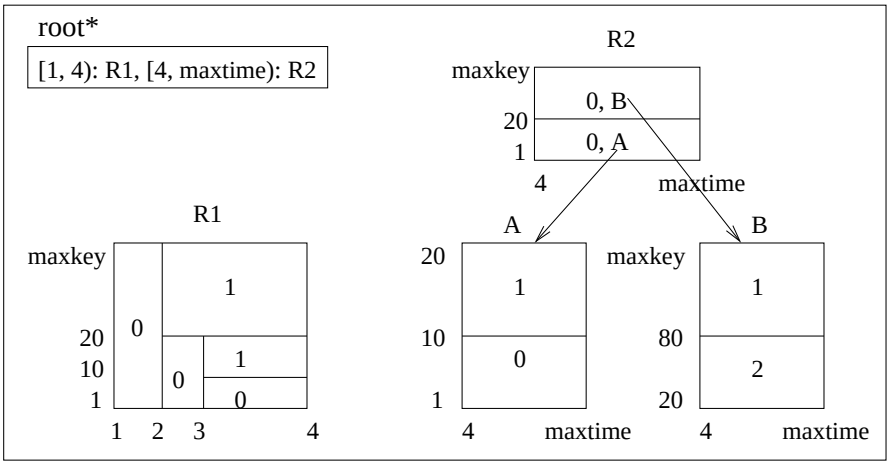
4.3.1 Split, strong condition, and $root^*$

With more objects inserted to the MVSB-tree with a single page, eventually the page overflows. To handle an overflow, all the alive records (whose rectangles end at *maxtime*) in the page are copied out to a new page. This is called a *time split*. After a time split, the newly generated page may be almost full. In such a case, a few subsequent insertions in the page trigger a time split again, resulting a space cost of $\Theta(1)$ block per insertion. To avoid this phenomenon, we require that after a time split, the new block should have at most $f \cdot b$ records, where constant $f \in (0, 1)$ is called the *strong factor*, and b is the maximum number of records a page can hold. We call this requirement the *strong condition*. If a newly generated page due to a time split *strong overflows* (having more than $f \cdot b$ records), it is *key split*, that is, it is split into two (or more, if f is small) by key and the records are distributed evenly among these pages.

As an example, consider the insertion of $\langle 4, 80\rangle : 1$ into Figure 9(a). Assume $b = 6$, and $f = 0.5$. The insertion causes the record whose rectangle contains (4, 80) to split (Figure 10(a)). Notice that here the new record located at the upper-right corner of space has value 1, according to the logical splitting optimization. Now the page has 7 records, while the maximum capacity is 6. So the page is overflowing. The four records touching the right border of space are copied out to a new page. The strong condition says that this new page should have no more than $f \cdot b = 3$ records. Since it is strong overflowing, a key split takes place which distributes the records evenly into two pages (Figure 10(b)), with key ranges [1, 20) and [20, *maxkey*), respectively. Notice that according to the logical splitting optimization, in the page with key range [20, *maxkey*), the value of the lowest record should be adjusted by adding to it the total value of records in the other page. An index page should then be allocated to reference the two new pages. This index page is the second root page of the index. Its start time is the current time (=4), while the previous root ends at time 4. All the root nodes are indexed by a structure called $root^*$. The MVSB-tree after the insertion is shown in Figure 10(c).



(a) inserting <4, 80>:1 leads to an overflow (b) strong overflow leads to a key split



(c) after inserting <4, 80>:1, there are two roots

Figure 10: Inserting $\langle 4, 80\rangle : 1$ to Figure 9(a).

4.3.2 Update an index entry, instead of updating all leaf entries in the sub-tree

An additional insertion will introduce the update of index entries. Consider the insertion of $\langle 5, 10\rangle : -1$ into Figure 10(c). The insertion goes to the alive root R_2. In the root page, there are two alive index entries pointing to page A and B, respectively. The key range of A contains the key=10 to be inserted. So the insertion recursively goes to the page A. The key range of B, which is [20, $maxkey$), is fully covered by the range [10, $maxkey$). Instead of updating all records in page B, we update the index entry pointing to page B. The result of the update is shown in Figure 11.

4.3.3 At any time instant, the MVSB-tree is an SB-tree on the key space

To see that the MVSB-tree uses the SB-tree structure on the key space, let's assume the time dimension collapses to a single time instant. The dominance-sum problem in this 1-dimensional space becomes:

Consider a set S of 1-dim objects, where every object $o \in S$ has a key $o.key$ and a value $o.value$. Given a query key q, compute the total value of objects in S whose keys $\leq q$.

Notice that $\forall$ object $o \in S$, $o.key \leq q$ if and only if $q \in$ [o,maxkey). So the 1-dim dominance-sum problem can be transformed to:

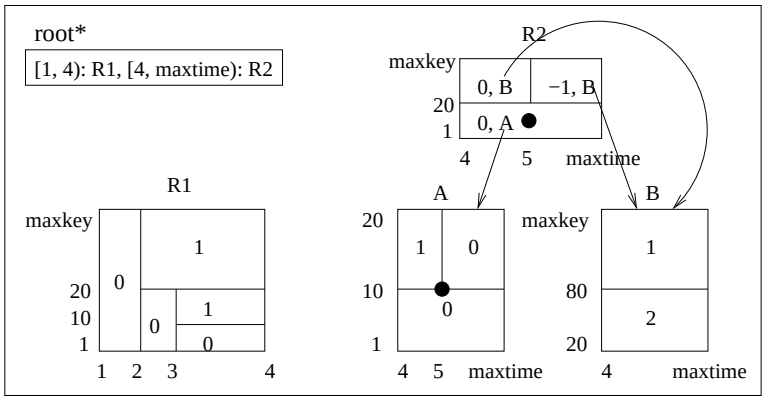


Figure 11: Inserting $\langle 5, 10\rangle : -1$ to Figure 10(c).

Consider a set S of interval objects, where every object $o \in S$ has an interval [o.key, maxkey) and a value o.value. Given a query key q, compute the total value of objects whose intervals contain q.

This latter problem is exactly the problem the SB-tree was proposed to solve! In other words, the SB-tree perfectly solves the 1-dim dominance-sum problem, if we ignore the time dimension. To extend the solution to involve the time dimension, a natural extension is to make an SB-tree partially persistent. Logically, the partially-persistent SB-tree (also called Multi-version SB-tree) is equivalent to a series of SB-trees, one at each time instant. An insertion operation and a point query involving time t are directed to the SB-tree corresponding to t. Physically, of course, it is too expensive to store a separate SB-tree at every time instant. The features from the MVBT can be applied to reduce the space. While logically equivalent to a set of B+-trees, one at each time instant, the MVBT nicely embeds the set of B+-trees in such a way that the overall space is linear [BGO$^+$96]. The structure we propose is named the MVSB-tree, because it is indeed a multi-version SB-tree index.

4.3.4 The structure

The MVSB-tree is a directed acyclic graph of disk-resident nodes that results from incremental insertions to an initially empty SB-tree. It has a number of SB-tree root nodes that partition the time space in such a way that each SB-tree root stands for a disjoint time interval and the union of these intervals covers the whole time space. A point query for a certain time instant t is directed to the root node whose time interval contains t. References to the root nodes are maintained in a structure called *root** which can be implemented as a B+-tree.

There are two types of pages in a MVSB-tree: the *index pages* and the *leaf pages*, all having the same size. An index page contains routers pointing to child pages, while a leaf page does not. For simplicity, we assume that both a leaf page and an index page have the same *maximum capacity* of b records. A *leaf record* (one stored in a leaf page) has the form $\langle range, interval, value\rangle$ where *range*, *interval* gives a rectangle in the key-time space and *value* is an aggregate value which is associated with every point in the rectangle. An *index record* (one stored in an index page) has the form $\langle range, interval, value, child\rangle$. Compared with a leaf record, it has a router pointing to some child page. Each page p also has a rectangle, where *p.range* is the union of the ranges of all the records in the page and *p.interval* is the time interval between the time the page is created and the time the page is copied. A page is said to be alive if it has not been copied yet. The following

property shows the relationships among the records in a page:

Property 1 *All the records in a MVSB-tree page have non-intersecting rectangles whose union is equal to this page's rectangle.*

Since we assume that insertions come in non-decreasing time order, an insertion only goes into an alive page and it only affects the alive records in the page. Consider an alive page p and all the alive records in p. Due to property 1, the key ranges of these records do not intersect and their union is equal to $p.range$. For ease of discussion, we define some terms regarding the alive records in p. Given a key $k \in p.range$, a *partly-covered record* is one whose key range intersects with, but is not contained in, $[k, maxkey)$; a *fully-covered record* is one whose key range is contained in $[k, maxkey)$; a *first fully-covered record* is a fully-covered record whose key range is lower than that of any other fully-covered record. Obviously, for any key $k \in p.range$, there can be at most one partly-covered record and at most one first fully-covered record. If p is an index page, we also call the child page which is pointed to by the partly-covered record as the *partly-covered child* page.

The Logical Splitting Optimization, page split, and strong condition we discussed before for a leaf page all apply to the index page. These concepts will be clearer after studying the insertion algorithm.

4.4 Detailed Algorithms

This section formally describes the insertion and point query algorithms for the MVSB-tree.

Algorithm *PointQuery*(Key k, Time t)

1. Find the root page p which is alive at t;
2. Return PagePointQuery(p, k, t).

Algorithm *PagePointQuery*(Page p, Key k, Time t)

1. $v = 0$;
2. for every record rec in p do
3. if rec is alive at t and $rec.low \leq k$ then
4. $v = v + rec.value$;
5. endif
6. endfor
7. if p is a leaf page then
8. return v;
9. else
10. Find the record rec whose rectangle contains $\langle k, t \rangle$;
11. return v + PagePointQuery($rec.child, k, t$);
12. endif

Algorithm *Insert*(Key k, Time t, Value v)

1. // Find the path of nodes containing partly covered records
2. $level = 0$;
3. $lowestpage$ = ReadPage(the latest root);
4. while $lowestpage$ is an index page and $lowestpage$ contains a partly-covered record $irec$, do

```
5.      path[level] = lowestpage;
6.      level++;
7.      lowestpage = ReadPage(irec.child);
8.  endwhile
9.  // Handle lowestpage
10. if lowestpage is a leaf page then
11.     if lowestpage has enough space then
12.         if there is a partly-covered record then
13.             Split it in lowestpage;
14.         else
15.             Split in lowestpage the first fully-covered record;
16.         endif
17.     else
18.         Copy alive leaf records from lowestpage to buffer;
19.         if there is a partly-covered record then
20.             Split it in buffer;
21.         else
22.             Add v to the first fully-covered record in buffer;
23.         endif
24.     Create new leaf pages (from records in buffer) and store their references in toparent;
25.     endif
26. else // lowestpage is an index page
27.     // similar to the leaf page case; omit.
28. endif
29. // Handle the pages which contain partly-covered
    // records bottom-up
30. for x = level − 1 downto 0 do
31.     if path[x] has enough space then
32.         if toparent is not empty then
33.             Insert records from toparent to path[x];
34.         endif
35.         Split the first fully-covered record in path[x], if any;
36.     else
37.         Copy alive records from path[x] to buffer;
38.     Add v to the first fully-covered record in buffer, if any;
39.         Copy toparent to buffer if it is not empty;
40.     Create new index pages (from records in buffer) and store their references in toparent;
41.     endif
42. endfor
43. // Decide whether to create a new root page
44. if toparent is not empty then
45.     Create a new root page from records in toparent;
46. endif
```

4.5 The Record Merging Optimization and the Page Disposal Optimization

In this section we describe two additional optimization techniques (besides the Logical Splitting Optimization) which should be integrated to the MVSB-tree.

4.5.1 The Record Merging Optimization

Record merging, if applicable allows to compact more records in a page and thus leads to less overall space. Two leaf records $lrec_1$, $lrec_2$ in the same page can be merged either horizontally (*time merge*) or vertically (*key merge*). A time merge can take place if (a) $lrec_1.range = lrec_2.range$; (b) $lrec_1.end = lrec_2.start$; and (c) $lrec_1.value = lrec_2.value$ (Figure 12(a)). A key merge can take place if (a) $lrec_1.interval = lrec_2.interval$; (b) $lrec_1.high = lrec_2.low$; and (c) $lrec_2.\ value = 0$ (Figure 12(b)).

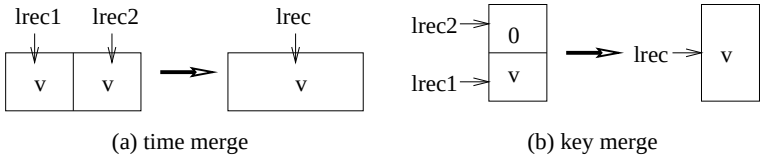


Figure 12: Time merge and key merge of two records.

The index records can be merged similarly. The difference of merging index records from merging leaf records is that two index records can be merged only if they point to the same child page.

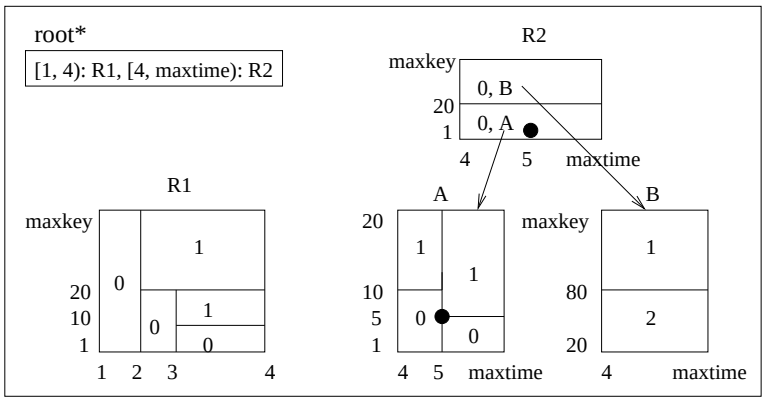


Figure 13: Inserting $\langle 5, 5\rangle : 1$ to Figure 11.

To see an example of time-merge, consider inserting $\langle 5, 5\rangle : 1$ to Figure 11. The two index entries in R_2 that both point to B will be time merged. In the same example, a key merge happens in page A. The MVSB-tree after the insertion is shown in Figure 13.

4.5.2 The Page Disposal Optimization

Since we allow many insertions at the same time instant we should update the index about the "net" effect of these insertions. However, our algorithms process one update at a time. Hence we introduce the page-disposal optimization, which spares the index from "intermediate" results. If a page which

is created at time t takes some subsequent insertions also at t and overflows, after the page is time split and key split, the page itself as well as the index record pointing to it can be physically removed from the index. This optimization saves space, too.

4.6 Complexity Analysis

For ease of discussion, we assume the record merging and the page disposal optimizations are not applied. Though these techniques improve performance, the worst-case bounds presented in the following also hold without applying the techniques. Let us discuss the impact of the strong factor f. Due to the strong condition, there are at most $f \cdot b$ alive records in a page that has been created. In order to guarantee a fan-out of at least 2, f has to be greater than $\frac{3}{b}$.

If a page overflows, the max number of new pages to be generated is given in lemma 1.

Lemma 1 *If a page overflows, the time split and possible key split will generate at most* $\lceil \frac{1.5}{f} + \frac{1}{3} \rceil$ *new pages.*

Proof. If a leaf page overflows, the max number of alive records to be copied is $b+1$. So the max number of newly generated pages is $\lceil \frac{b+1}{f \cdot b} \rceil$. Since $f \cdot b \geq 3$, $\lceil \frac{b+1}{f \cdot b} \rceil \leq \lceil \frac{1}{f} + \frac{1}{3} \rceil \leq \lceil \frac{1.5}{f} + \frac{1}{3} \rceil$. Suppose the lemma is true for all the child pages of an index page p. If p overflows, the max number of alive records to be copied is $b + \lceil \frac{1.5}{f} + \frac{1}{3} \rceil - 1$. So the max number of newly generated pages is given by $\lceil \frac{b+\lceil \frac{1.5}{f} + \frac{1}{3} \rceil - 1}{f \cdot b} \rceil \leq \lceil \frac{1}{f} + \frac{1}{3} \cdot (\frac{1.5}{f} + \frac{1}{3}) \rceil \leq \lceil \frac{1.5}{f} + \frac{1}{3} \rceil$. □

After a page p is created and before it is copied, the effect of an insertion in p may be the addition of some new records and the logical deletion of some others. The amount of additions and logical deletions are bounded as shown in lemma 2.

Lemma 2 *An insertion in an alive page* p *which does not overflow introduces at most* $\lceil \frac{1.5}{f} + \frac{4}{3} \rceil$ *additions and at most 2 logical deletions.*

Proof. The reason why there are at most 2 logical deletions is straightforward: For a leaf page, there is only one record to be logically deleted. This is the partly-covered record (if there is one) or the first fully-covered record (otherwise). For an index page, there can be 0, 1 or 2 logical deletions: If the partly-covered child page is time split, the partly-covered record is logically deleted; if there is any fully-covered record, the first fully-covered one is also logically deleted.

We now focus on additions. For a leaf page, there can be 1 or 2 additions (1 for a fully-covered record and 2 for a partly-covered one). Since $2 \leq \lceil \frac{1.5}{f} + \frac{4}{3} \rceil$, the lemma is correct for a leaf page. For an index page, the possible additions are from splitting the first fully-covered record and from the time split (and then key split) of the partly-covered child page. The maximum number of additions from splitting the first fully-covered record is 1. The maximum number of additions from splitting the partly-covered child page is $\lceil \frac{1.5}{f} + \frac{1}{3} \rceil$ (lemma 1). The total additions is thus at most $\lceil \frac{1.5}{f} + \frac{4}{3} \rceil$. □

For any time t during the lifespan of a page p, it is guaranteed that there is at least a certain number of records in p which are alive at t, as shown in lemma 3.

Lemma 3 *Given time* t*, any page* p *which is alive at* t *(except the root) contains at least* $\lceil \frac{f \cdot b}{2} \rceil$ *records alive at* t*.*

Proof. Let $p_1, p_2, \cdots, p_x$ be the longest successor path to p, i.e. $\forall i \in [1, x-1], p_{i+1}$ is a successor of p_i and $p_x = p$. Since p is not a root page, somewhere in the path there must be a key split. Let p_i be

the result of the last key split which occur in the path. Suppose when p_i was about to be generated, there were $x \cdot f \cdot b - y$ records, where $x \geq 2$ and $0 \leq y < f \cdot b$. Right after p_i was generated, the number of records in it is at least $\lfloor \frac{x \cdot f \cdot b - y}{x} \rfloor = \lfloor f \cdot b - \frac{y}{x} \rfloor \geq \lfloor f \cdot b - \frac{f \cdot b - 1}{2} \rfloor = \lceil \frac{f \cdot b}{2} \rceil$.
Since in a page, the number of additions is no smaller than the number of deletions, for any time $t1$ before $p_i.end$, there are at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at $t1$. For all $j \in [i+1, x]$, when p_j is created, it has at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at $t1$ since there were at least this many to be copied from p_{j-1} and there is no strong overflow. For any later time before $p_j.end$, the number of alive records does not decrease. □

Suppose K is the number of different keys ever inserted into the MVSB-tree. Lemma 4 gives the upper bound of the height of a MVSB-tree in regards to K.

Lemma 4 *The upper bound of the height of any sub-tree in a MVSB-tree is* $\lceil \log_{\lceil \frac{f \cdot b}{2} \rceil} (K+1) \rceil$.

Proof. Given a tree in an MVSB-tree. Consider each time instant $t \in$ the lifespan of the tree root. Since there are at most K different keys ever inserted in the tree, there are at most $K+1$ different leaf records which are alive at t. Since each leaf page alive at t contains at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at t, there are at most $\frac{K+1}{\lceil \frac{f \cdot b}{2} \rceil}$ leaf pages alive at t. This also means that there are at most this many index records which are alive at t and which point to these pages. So at one level up, there are at most $\frac{K+1}{\lceil \frac{f \cdot b}{2} \rceil^2}$ index pages alive at t. This argument is true for all levels until the root, where there is only one page alive at t. So there are at most $\lceil \log_{\lceil \frac{f \cdot b}{2} \rceil} (K+1) \rceil + 1$ levels. □

Suppose there are n insertions in a MVSB-tree. Theorems 3 states the worst-case insertion cost, point query cost and the space complexity, respectively.

Theorem 3 *For a MVSB-tree, the number of disk page accesses is* $O(\log_b K)$ *for an insertion and* $O(\log_b n)$ *for a point query. The space complexity is* $O(\frac{n}{b} \cdot \log_b K)$.

Proof. First, we examine the worst case insertion cost. An insertion operation first traverses the tree from the latest root page to a leaf page, then traverses back, requiring constant number of I/Os per node along the path. Since the tree height is $\lceil \log_{\lceil \frac{f \cdot b}{2} \rceil} (K+1) \rceil = O(\log_b K)$, an insertion needs $O(\log_b K)$ I/Os.

Second, we examine the cost of a point query. If the root page which is alive at the query time instant is found, it takes $O(\log_b K)$ I/Os to answer a point query in the worst case. If the *root** is kept as a B+-tree, extra I/Os are needed to locate the root. Since after a root page is generated, it takes at least $O(b)$ insertions for it to overflow (lemma 2), there are $O(n/b)$ root pages. So it takes $O(\log_b n)$ to locate the root in the worst case. To sum up, a point query needs $O(\log_b n)$ I/Os in the worst case.

Last, we examine the worst case space complexity. We consider the total number of occupied slots in all the SB-trees embedded in the MVSB-tree (if a record is copied, the two copies are considered to occupy different slots). We will show that each insertion creates $O(\log_b K)$ new occupied slots. We partition the occupied slots into two sets: in the first set, the occupied slots are created from copying existing occupied slots; the rest are in the second set. Each insertion creates $O(\log_b K)$ slots in the second set (lemma 2).

For the first set: We know that after a page is created, it takes at least $O(b)$ insertions for it to overflow (lemma 2). So when a page overflows, there were at least $O(b)$ insertions that went through this page after it was created. On the other hand, the overflow introduces at most $O(b)$ occupied

slots in the first set. So we can amortize the O(b) occupied slots to the O(b) insertions. Thus each insertion creates O(1) amortized copied slot for each page it goes through. Since an insertion goes through at most O($\log_b K$) pages, an insertion creates O($\log_b K$) slots in the first set as well.

To sum up, each insertion creates O($\log_b K$) occupied slots. So for n insertions the total number of occupied slots is O($n \cdot \log_b K$). Now we consider the minimum occupancy of a page. Each non-root page has at least $\lceil \frac{f \cdot b}{2} \rceil = O(b)$ occupied slots (lemma 3). Clearly, except for the last root, all the root nodes have a minimum occupancy of $O(b)$, too. So the total number of pages occupied by the SB-trees in an MVSB-tree is O($\frac{n}{b} \cdot \log_b K$).

Now we consider the space occupied by the *root**, if it is kept in a B+-tree. Since there can be at most O(n/b) roots, the space occupied by the B+-tree is O(n/b^2). To add up, the overall space of the MVSB-tree is O($\frac{n}{b} \cdot \log_b K$). □

A corollary of theorems 2 and 3 summarizes the performance of maintaining and computing the range-temporal aggregates as follows.

Corollary 1 *Using two MVSB-trees, a plain range-temporal aggregation query is answered in O($\log_b n$) I/Os. The update cost is O($\log_b K$) while the space complexity is O($\frac{n}{b} \cdot \log_b K$).*

The O($\log_b n$) in the range-temporal aggregation query time is due to the time needed identifying the root of the appropriate SB-tree in the MVSB-tree graph. In practice, this search can be even faster if all different SB-tree roots created in the evolution are kept in a main-memory array, in which case the query time is reduced to traversing the appropriate SB-tree, i.e., O($\log_b K$).

5 The Functional Range-Temporal Aggregation

Section 5.1 reduces the functional range-temporal aggregation problem into a special case. Section 5.2 then proposes a solution to the special case, which in turn can solve the general case. Finally, Section 5.4 extends the solution to the case when a record has a value function instead of a constant value.

5.1 Reduction to Origin-Involved Special Cases

To support functional range-temporal aggregation, we only need to support the special case where the lower-left corner of the query rectangle is the lower-left corner of the space (called *origin*).

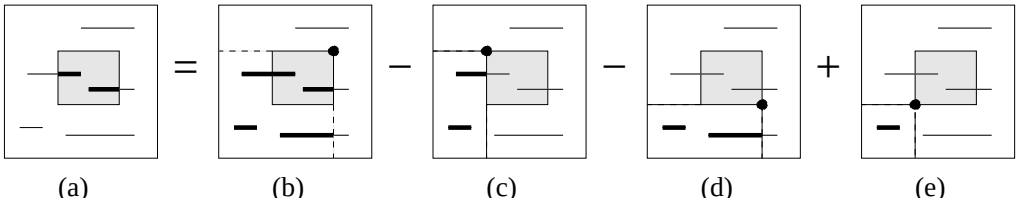


Figure 14: Reduction from the functional range-temporal aggregation to the origin-involved special case.

Figure 14(a) illustrates an arbitrary functional range-temporal aggregation query. The query rectangle is shown as a shadowed box. The five temporal records are shown as the horizontal line segments.

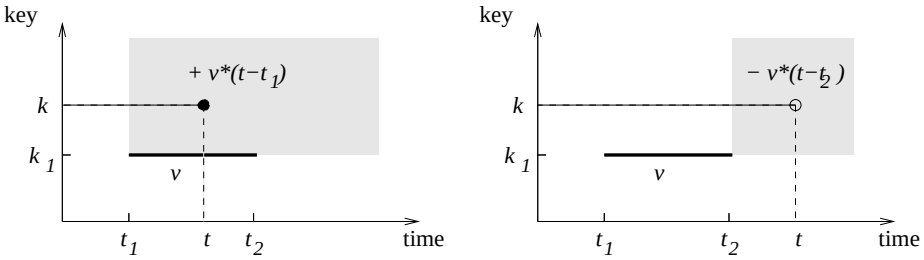
There are two records intersecting the query rectangle, and the functional range temporal aggregation query computes the total value of them. The contribution of each record to the query, as illustrated by a thick line segment, is the value of the record multiplied by the length of its intersection with the query rectangle.

Figure 14(b) illustrates another functional range-temporal aggregation query. The query rectangle is cornered by the origin of space and the upper-right corner of the previous shadowed query rectangle. This query is a special case since the query rectangle contains the origin of space. We thus call such a query the *origin-involved special case*. To specify an origin-involved query, we only need to give a single point in the key-time space: the upper-right corner of the query rectangle.

As shown in Figure 14, an arbitrary functional range temporal aggregation query can be reduced to the origin-involved special case. To compute an arbitrary aggregation (Figure 14(a)), we first compute the origin-involved aggregation regarding the upper-right corner (Figure 14(b)). This query gives a larger result value since the parts of records to the left and below the original query rectangle are also counted. To subtract those to the left of the query rectangle, another origin-involved aggregation (Figure 14(c)) is performed. Similarly, we subtract those below the query rectangle (Figure 14(d)). Finally, the records both to the left and below the query rectangle (Figure 14(e)) are subtracted twice, and thus they need to be added back.

5.2 Solution to the Origin-Involved Functional Range-Temporal Aggregation Problem

Our methodology of computing the origin-involved functional aggregates is as follows. Given a set of temporal records, for every point p in the two-dimensional key-time space, the origin-involved functional aggregation regarding p is a single value. We design an index structure which logically maintains such values for all points in the key-time space. As the set of records are dynamically updated, this hypothetical index is updated accordingly. To compute an origin-involved aggregate, we perform a point query on this index. To implement our methodology, it remains to decide: (1) how the updates of the set of temporal records are reflected on the index; and (2) how to perform a point query.



(a) Effect of insertion (b) Effect of deletion

Figure 15: Effect on the origin-involved aggregation index upon insertion/deletion of a temporal record.

We first discuss the transformation of updates. Note that we consider the transaction time model, and thus the updates to the original set of temporal records are only insertions and deletions. For instance, the record in Figure 15(a) with key=k_1, time interval=$[t_1,\ t_2]$, and value v corresponds to two updates: an insertion at t_1 and a deletion at t_2. As Figure 15(a) shows, to insert at t_1, the

hypothetical origin-involved aggregation index is affected as follows: The value of each point $(k,\ t)$ in $[k_1,\ maxkey)$x$[t_1,\ maxtime)$ should be increased by

$$v * (t - t_1) = v * t - v * t_1.$$

Similarly, as Figure 15(b) shows, to delete the temporal record at t_2, the origin-involved index is affected as follows: The value of each point $(k,\ t)$ in $[k_1,\ maxkey)$x$[t_2,\ maxtime)$ should be decreased by $v * (t - t_2)$. Or equivalently, be increased by

$$-v * t + v * t_2.$$

Note that such updates have exactly the same format as the update of a MVSB-tree! The only difference is that in the MVSB-tree we presented in section 4, each update takes a constant value, while here, each update takes a linear function over time. Such a function can be stored in constant space, and can be added with another function, by storing and adding the coefficients, respectively. Therefore we can use the MVSB-tree to support the origin-involved functional temporal aggregation, with a slight modification that every aggregated "value" that is stored in the index is a pair of values instead of one.

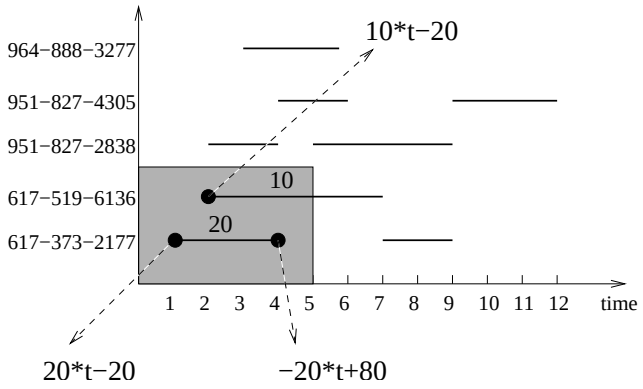


Figure 16: Computing an origin-involved functional range-temporal aggregate. The dominance-sum is $10 * t + 40$, which evaluates to 90 at $t = 5$.

As an example, in Figure 16 the functions associated with three end points of objects are shown – the three end points dominated by the upper-right corner of the shadowed query region. All end points are inserted to an MVSB-tree, where the "value" associated with each end point is the associated function, or the two coefficients of the function. For instance, the three end points are associated with "values" (20, -20), (-20, 80), and (10, -20).

Let's study how such built MVSB-tree index can help us evaluate the functional range-temporal aggregation query. Intuitively, the two phone call records, with values 10 & 20 per unit of time, respectively, both lasted three units of time in the query region. Therefore the functional range-temporal aggregation result should be 10*3+20*3=90. Let's see how the dominance-sum index (MVSB-tree) can compute it.

The MVSB-tree can help us compute the dominance-sum. In this case, since the upper-right corner of the query region dominates three end points, the dominance-sum is the "total value" of (20, -20), (-20, 80), and (10, -20). This total value is (20-20+10, -20+80-20)=(10,40), which corresponds to a function $10t + 40$. At time 5, the function evaluates to be 10*5+40=90, which matches our intuition.

5.3 The Overall Picture

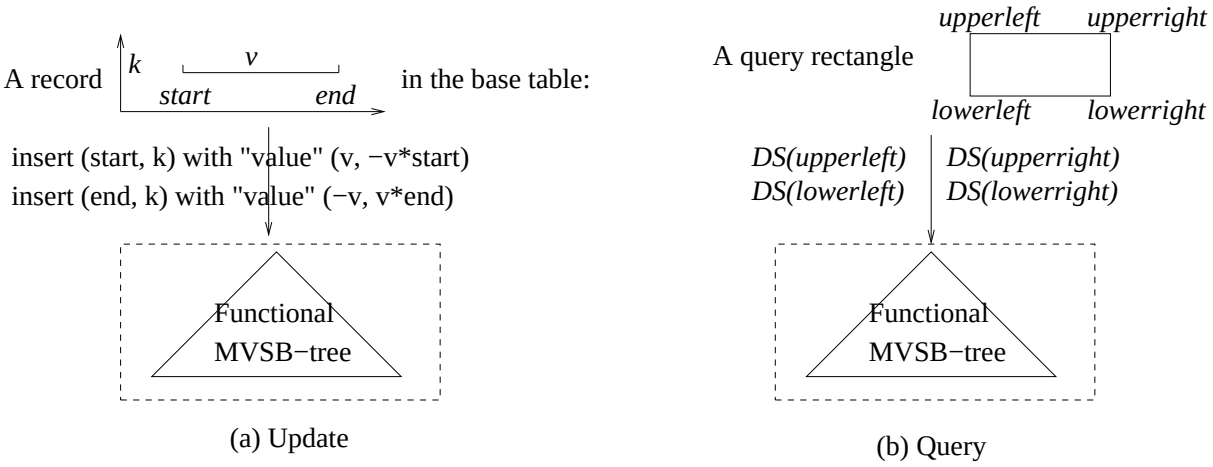


Figure 17: Illustrating of the functional range-temporal aggregation scheme.

The insertion and query schemes supporting the functional range temporal aggregation query, utilizing the MVSB-tree as basic building block, are illustrated in Figure 17. Different from the plain range temporal aggregation case, here a single MVSB-tree is needed. Both the start point and end point of a record in the base table are inserted to this MVSB-tree. As illustrated in Figure 14, a functional range-temporal aggregation query result is computed as:

$$DS(upperright) - DS(upperleft) - DS(lowerright) + DS(lowerleft)$$

Here each dominance-sum $DS(\cdot)$ is first computed as a value function, then evaluated at the time component of the query point.

5.4 Extension to Support Value Functions

So far we have discussed the case when the value for every record is a constant. In fact in the functional aggregation case, we actually treated it as a constant function. In general, a record may have a non-constant value function. Our technique still works, provided that some conditions hold.

Here are the extensions:

To insert (at t_1) a record with key=k_1, time interval=$[t_1,\ t_2]$, and value function=$f(t)$, the value of each point $(k,\ t)$ in $[k_1,\ maxkey)$x$[t_1,\ maxtime)$ should be increased by

$$h_1(t) = \int_{t_1}^{t} f(x)dx$$

And to delete (at t_2) the record, the value of each point $(k,\ t)$ in $[k_1,\ maxkey)$x$[t_2,\ maxtime)$ should be decreased by

$$h_2(t) = \int_{t_2}^{t} f(x)dx$$

An arbitrary value function may not suit the need for functional aggregation. For instance, a function, which cannot be represented in constant space, or whose integral cannot be computed, does not satisfy

our needs. We hereby propose six requirements as guidelines of choosing value functions which suit this need.

1. The function should have a fixed format.
2. The function should be represented in constant space.
3. The summation of two functions should be easily performed, and the result should have the same format.
4. The negation of the function should be easily performed, and the result should have the same format.
5. The function should have efficient evaluation.
6. the integral of the function should be able to be efficiently computed, and should satisfy the above five requirements.

Note that if $f(x)$ is a polynomial function of rank a, function $h_1(t)$, $h_2(t)$ are polynomial functions of of rank $a+1$. For example, if $f(x) = 0.5x - 0.5$, then $h_1(t) = \int_{t_1}^{t}(0.5x - 0.5)dx = 0.25t^2 - 0.5t + (0.5t_1 - 0.25t_1^2)$. Such functions have fixed format ($c_1 * t^2 + c_2 * t + c_3$, where $c_1,\ c_2,\ c_3$ are constants), can be represented in constant space (by storing their coefficients), can be added up or negated (by manipulating on their coefficients), and can be evaluated efficiently. This means our solution for the continuous case of the functional aggregation works for polynomial temporal record functions, which is a rather general class of functions, covering many practical applications.

6 Performance Results

This section provides experimental results evaluating the MVSB-tree based approaches for the plain and functional range-temporal aggregation query. The experimental setup is given in Section 6.1. Section 6.2 compares the query performance of the MVSB-tree based approaches against the naive approach of retrieving the records satisfying the range-interval condition and aggregating their values on the fly. The range-interval selection in the naive approach is performed by querying a traditional temporal index, the MVBT [BGO$^+$96]. Section 6.3 reveals how good the three optimization techniques are. Finally, Section 6.4 shows the performance of the functional temporal aggregation query.

6.1 Experimental Setup

The algorithms were all implemented in C++ using GNU compilers. All experiments are performed on a Dell Pentium IV 3.2GHz PC with 1GB memory. Unless otherwise stated, the experiments use the following default parameters: page size is 4KB, an LRU buffer with 64 pages, strong factor $f = 0.9$.

For the query performance, we measure the average execution time of 100 randomly generated query rectangles with fixed rectangle shape and size. The shape of a rectangle is described by the *R/I ratio*, where R is the length of the query key range divided by the length of the key space and I is the length of the query time interval divided by the length of the time space. The *query rectangle*

size (QRS) is described by the percentage of the area of the query rectangle in the whole key-time space. The default value of R/I ratio is 1, and the default value of QRS is 1%.

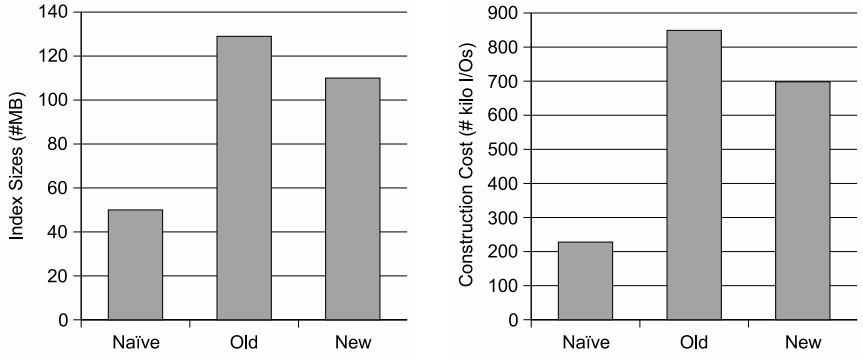
The dataset used in the experiments has 1 million time intervals generated using the Time-IT software [KS98]. Here the time space is [1, 10^8). Note that the Time-IT software does not generate record keys. We add keys by first generating 10,000 random keys in the key space [1, 10^6), then assigning the time intervals to the keys in a round-robin fashion. Each key corresponds to 100 intervals. The $key, start, end, value$ attributes of each record are all 4 bytes long.

6.2 Comparison with the Naive Approach

This section compares the generation time, index size, and query performance of three algorithms:

- **Naive**: the naive approach of using an range-interval selection query on an MVBT index to find the actual records and then aggregate their values on the fly.
- **Old**: the MVSB-tree-based approach using the old reduction technique, which reduces a plain range-temporal aggregation query to six LKLT and/or LKST queries.
- **New**: the MVSB-tree-based approach using the new reduction technique, which reduces a plain range-temporal aggregation query to four dominance-sum queries.

The goals of comparing these three algorithm are two-fold. First, we want to show how much faster our proposed approach (based on specialized aggregation index) is over the naive approach (based on object retrievals). Second, we want to compare the two versions of our new approach, using the old and new reduction techniques, respectively.



(a) Index size (b) Construction cost

Figure 18: Comparison of index size and construction cost against the naive approach.

First, we compare the index size and index generation cost of the three approaches. As Figure 18 shows, the MVSB-tree-based solution occupies more space and takes longer to generate. This is to be expected, since the both MVSB-tree based approaches use two MVSB-trees, and each MVSB-tree has a $O(\log_b K)$ overhead in worst-case asymptotic space cost.

Comparing the two MVSB-tree-based solutions, the *New* solution is more efficient both in space cost and in construction cost. The reason is that the *New* approach needs to perform two updates per time interval, while the *Old* approach needs to perform three updates per time interval.

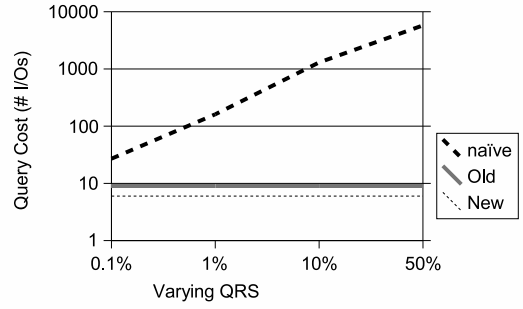
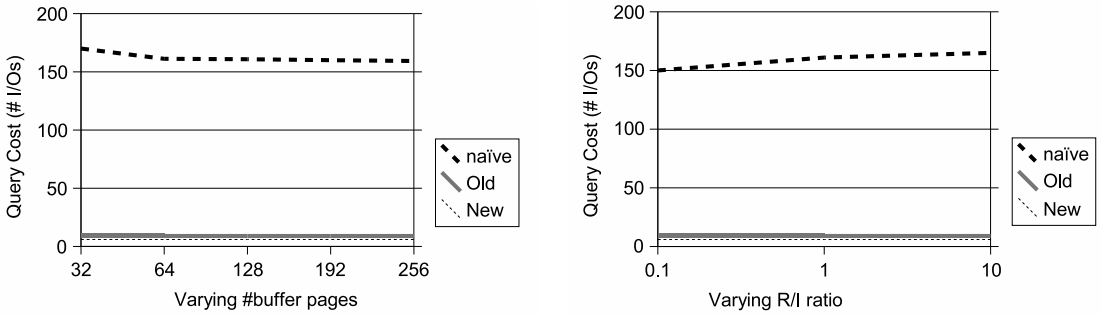


Figure 19: Query performance comparison, varying QRS, against the naive approach.

Figure 19 shows that both versions of our approach are multiple orders of magnitude faster than the naive approach in terms of query performance. Here the Y axis is in logarithmic scale. The larger the QRS is, the more advantageous our approach is. This is to be expected, since the MVSB-tree-based query algorithm has logarithmic complexity, independent to the QRS, while the naive approach has linear complexity and in the worst case scans the whole database.

Between these two versions, the *New* approach is more efficient than the *Old* approach. The reason is that the *New* approach answers a plain range-temporal aggregation query by four dominance-sum queries, and the *Old* approach answers a plain range-temporal aggregation query by six dominance-sum queries.



(a) varying buffer size (b) varying R/I ratio

Figure 20: Query performance comparison, varying buffer size and R/I ratio, against the naive approach.

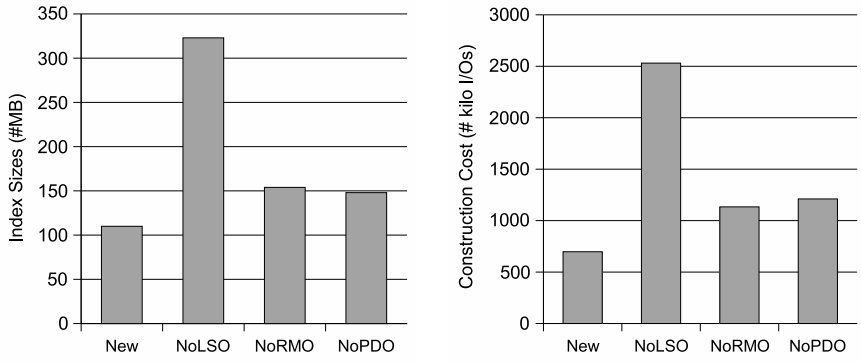
Figure 20(a) and (b) compares the query performance of the three algorithms, while varying the buffer size and R/I ratio, respectively. Here QRS is the default value (1% of the key-time space). Again, in both cases the *New* approach is slightly more efficient than the *Old* approach, and both these two MVSB-tree-based approaches are clearly superior than the naive approach.

In the remaining sections, we focus on evaluating the *New* algorithm alone.

6.3 Effect of the Optimizations

This section evaluates how good each of the three optimization techniques is. In particular, the three optimization techniques are:

- **LSO**: Logical Splitting Optimization.
- **RMO**: Record Merging Optimization.
- **PDO**: Page Disposal Optimization.



(a) Index size (b) Construction cost

Figure 21: The impact different optimization techniques have on the index size and construction cost.

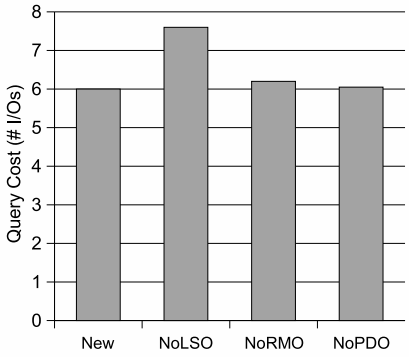
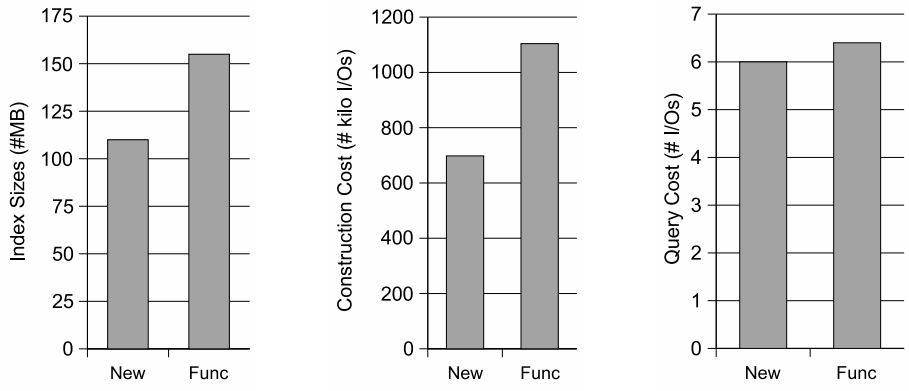


Figure 22: The impact different optimization techniques have on the query cost.

Figure 21(a), Figure 21(b), and Figure 22 demonstrate the impact the optimization techniques have on the index size, construction cost, and the query performance. Here **New** stands for the same algorithm we used in the previous section. That is, the MVSB-tree-based index solution for the plain range-temporal aggregation query utilizing the new reduction technique. **NoLSO** is the *New* solution without LSO. **NoRMO** is the *New* solution without RMO. And **NoPDO** is the *New* solution without PDO.

Clearly, out of the three optimization techniques, LSO is the most important one. Recall that the LSO enables the update algorithm to split a single entry in each page along the insertion path. Without it, in every page along the insertion path multiple entries need to split. Therefore the LSO optimization brings enormous savings in index size and update cost. With a more compact index, the query cost of the solution with LSO is also noticeably more efficient. The other two optimizations also have impact, but the impact is not as big.

6.4 Evaluation of the Functional Range Temporal Aggregation Query



(a) Index size (b) Construction cost (c) Query cost

Figure 23: Evaluation of the functional temporal aggregation query solution using the plain temporal aggregation query solution as baseline.

Figure 23 shows the performance result of the functional temporal aggregation solution. As baseline for comparison, we use the data of the plain temporal aggregation solution.

There are two differences in the index utilization between the two MVSB-tree-based indices for the functional case and the plain case.

1. In the functional case, a value associated with an interval record and an aggregated value associated with an index entry are both two numbers instead of one. The two numbers are the coefficients of the corresponding (aggregated) value function.

2. In the functional case, a single MVSB-tree is needed instead of two. Recall that the plain case keeps two MVSB-trees, one corresponding the start times of the records and the other corresponding to the end times of the records. One record generates two updates, one in each MVSB-tree. Here in the functional case, a single MVSB-tree is maintained. A record still corresponds to two updates, both in the same MVSB-tree.

The first difference indicates that the functional case should occupy more space. But the second difference suggests that the functional case may have less space requirement. This is because maintaining an MVSB-tree index has some overhead, e.g. the root structure which references all the root nodes, and the functional case saves on such overhead by maintaining one index instead of two.

The net effect, as shown in Figure 23(a), is that the functional case uses a little more space than the plain case. Consequently, update (Figure 23(a)) and query (Figure 23(b)) also are a little more expensive. But in all cases, the functional solution is not prohibitively expensive at all. So it is quite a reasonable extension of the plain case.

7 Conclusions

Temporal aggregates have become predominant operators in analyzing historical data. This paper examined temporal aggregation queries in the presence of key-range predicates. Such queries allow

the warehouse managers to focus on tuples grouped by some key range over a given time interval. We considered both plain and functional range-temporal aggregations. These problems are reduced to dominance-sum queries. We proposed a new index structure, the Multiversion SB-Tree (MVSB-tree), for incrementally maintaining and efficiently computing the dominance-sum queries and in turn the range-temporal aggregation queries. The MVSB-tree has very fast (logarithmic) query time and update time, at the expense of a small space overhead. The benefits of our solution were verified through experimental evaluation.

References

[BGJ06a] M. H. Böhlen, J. Gamper, and C. S. Jensen. How Would You Like to Aggregate Your Temporal Data. In *Proceedings of International Symposium on Temporal Representation and Reasoning (TIME)*, pages 121–136, 2006.

[BGJ06b] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional Aggregation for Temporal Data. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 257–275, 2006.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4):264–275, 1996.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

[BS96] J. van den Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 168–179, 1996.

[Cha88] B. Chazelle. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM Journal on Computing*, 17:427–462, 1988.

[GAA03] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-Tree: An Efficient Indexing Scheme for Range-Aggregate Queries. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 143–157, 2003.

[GAE00] S. Geffner, D. Agrawal, and A. El Abbadi. The Dynamic Data Cube. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 237–253, 2000.

[GGM+04] D. Gao, J. A. G. Gendrano, B. Moon, R. T. Snodgrass, M. Park, B. C. Huang, and J. M. Rodrigue. Main Memory-Based Algorithms for Efficient Parallel Aggregation for Temporal Databases. *Distributed and Parallel Databases*, 16(2):123–163, 2004.

[GHR+99] J. Gendrano, B. Huang, J. Rodrigue, B. Moon, and R. Snodgrass. Parallel Algorithms for Computing Temporal Aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 418–427, 1999.

[Jen98] C. S. Jensen et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*, pages 367–405. Springer, 1998.

[KCK04] S. T. Kang, Y. D. Chung, and M.-H. Kim. An Efficient Method for Temporal Aggregation with Range-Condition Attributes. *Journal of Information Sciences*, 168(1-4):243–265, 2004.

[KS95] N. Kline and R. Snodgrass. Computing Temporal Aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 222–231, 1995.

[KS98] N. Kline and M. Soo. Time-IT, the Time-Integrated Testbed. *ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz, Current as of August 18*, 1998.

[KTF98] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):1–20, 1998.

[LS89] D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*, pages 315–324, 1989.

[MLI00] B. Moon, I. Lopez, and V. Immanuel. Scalable Algorithms for Large Temporal Aggregation. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 145–154, 2000.

[PS85] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Sprinter Verlag, 1985.

[SS88] A. Segev and A. Shoshani. The Representation of a Temporal Data Model in the Relational Environment. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 39–61, 1988.

[ST99] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[TPF04] Y. Tao, D. Papadias, and C. Faloutsos. Approximate Temporal Aggregation. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 190–201, 2004.

[Tum92] P. Tuma. Implementing Historical Aggregates in TempIS. *Master's thesis, Wayne State University, Michigan*, 1992.

[YK97] X. Ye and J. Keane. Processing Temporal Aggregates in Parallel. In *Proc. of Int. Conf. on Systems, Man, and Cybernetics*, pages 1373–1378, 1997.

[YW01] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 51–60, 2001.

[YW03] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 12(3):262–283, 2003.

[ZMT+01] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 237–245, 2001.

[ZTG02] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient Aggregation over Objects with Extent. In *ACM International Symposium on Principles of Database Systems (PODS)*, pages 121–132, 2002.