On Computing Temporal Aggregates with Range Predicates

DONGHUI ZHANG
Northeastern University
ALEXANDER MARKOWETZ
Hong Kong University of Science and Technology
VASSILIS J. TSOTRAS
University of California, Riverside
DIMITRIOS GUNOPULOS
University of California, Riverside
and
BERNHARD SEEGER
Philipps Universität Marburg, Germany

Computing temporal aggregates 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 aggregate queries are critical. Previous approaches have aggregate predicates that involve only the time dimension. In this article 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. 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 aggregate 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 aggregates. Here, the value of any record is a

Authors' addresses: D. Zhang, College of Computer and Information Science, Northeastern University, 360 Huntington Av., #202WVH, Boston, MA 02115; email: donghui@cs.ucr.edu; A. Markowetz, Department of Computer Science and Engineering, University of Science and Technology, Clearwater Bay, Kowloon, Hong Kong, email: alexmar@cse.ust.hk; V. J. Tsotras, Department of Computer Science and Engineering, Bourns College of Engineering, University of California, Riverside, CA 92521, email: tsotras@cs.ucr.edu; D. Gunopulos, Department of Computer Science and Engineering, Bourns College of Engineering, University of California, Riverside, CA 92521; email: dg@cs.ucr.edu; B. Seeger, Fachbereich Mathematik & Informatik, Philipps Universität, Marburg, Germany, email: seeger@Mathematik.Uni-Marburg.de.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2008 ACM 0362-5915/2008/06-ART12 $5.00 DOI 10.1145/1366102.1366109 http://doi.acm.org/10.1145/1366102.1366109

function over time. The meaning of aggregates is altered such that the contribution of a record to the aggregate 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.

Categories and Subject Descriptors: H.4.0 [**Information Systems Applications**]: General

General Terms: Algorithms

Additional Key Words and Phrases: Temporal aggregates, indexing, range predicates, functional aggregates

ACM Reference Format:

Zhang, D., Markowetz, A., Tsotras, V. J., Gunopulos, D., and Seeger, B. 2008. On computing temporal aggregates with range predicates. ACM Trans. Datab. Syst. 33, 2, Article 12 (June 2008), 39 pages. DOI = 10.1145/1366102.1366109 http://doi.acm.org/10.1145/1366102.1366109

1. INTRODUCTION

With the rapid increase of historical data, temporal aggregates have become predominant operators for data analysis. Their computation is significantly more intricate than for traditional aggregation without a time dimension. Each data tuple is accompanied by a time interval, its so-called *lifespan*, during which its attribute values are valid. Similarly, queries are associated with a time interval. A temporal aggregate only considers tuples, whose lifespans intersect the query's time interval.

Consider the example of Figure 1. Each horizontal line segment represents a phone call, having a duration and a price. For instance, the record with a unit price of 35 cents started at time 4 and ended at time 6. The two records with right arrows are on-going call records, whose *end* time is not known yet. One may be interested in the total number of calls from the 951 area (Riverside, CA) during the time interval [3, 8]. This query, illustrated by the shadowed box, is an example of the plain range-temporal aggregate query. Specifically, the aggregate function is COUNT, and the result is 3 since there are three records intersecting the query range. Another widely used aggregate 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 SUM, where the value of every record is 1.

For some applications, one would want the contribution of a record to an aggregate to depend on the length of the intersection between the query's time interval and the record's lifespan. Again, consider the phone calls in Figure 1. Suppose the price associated with each record is the price per minute. The total cost of phone calls intersecting the query rectangle is $35 * 2 + 60 * 1 + 60 * 3 = 310$. In this case, the contribution of a record to the query is a function over time, for examples, $intersection * minuteprice$.

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:

—*A plain range-temporal aggregate query* computes the total value of records in S whose keys are in R and whose time intervals intersect I.

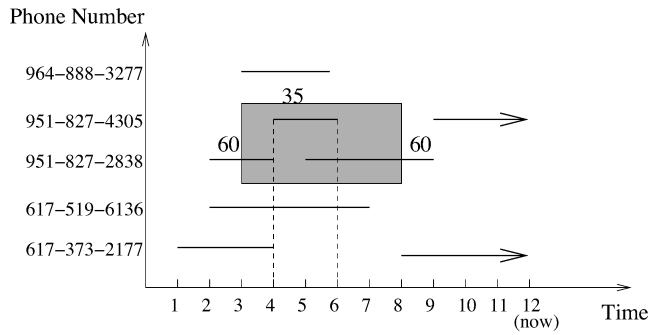


Fig. 1. An example range-temporal aggregate query.

—*A functional range-temporal aggregate query* computes the weighted total value of records in S whose keys are in R and whose time intervals intersect I. The weight on each record is proportional to the length of intersection between the record and I.

The two versions differ in how to explain the value of each object whose interval intersects the query rectangle (composed of R and I). But both of them are indeed temporal aggregate problems. In particular, the "plain range-temporal aggregate" problem should NOT be read as "plain-range temporal aggregate," which might suggest the query only considers a key range.

As assumed in many existing work in temporal databases, this article assumes the transaction-time model, where updates in the database happen in nondecreasing time order. We allow multiple updates to happen at the same time. When a temporal record with key $= k$ and value $= v$ starts at time t_1, an insertion is issued in the form of $ins\ (t_1, k) : v$. The *end* time of the record is stored as a special symbol *now*. When the record ends at time t_2, a logical deletion is issued in the form of $del\ (t_2, k) : v$. The correct *end* time of the record is recorded.

Range-temporal aggregate computation is closely related to the *selection queries*, which retrieve records intersecting the query range. A large number of temporal indices have been proposed to support the selection queries, as extensively surveyed in Salzberg and Tsotras [1999]. A straightforward approach to solve the range-temporal aggregate queries is to utilize such an index to find records intersecting the query range, and then aggregate their values on the fly. A major drawback however, is that computational efforts are proportional to the number of selected records. If many records satisfy the selection condition, the query performance can be as poor as a linear search. For many applications, for example, interactive analysis and decision making, such performance is prohibitive. Although there is a large volume of work on temporal aggregation, as reviewed in Section 2, they either do not solve the same problem, or do not solve the problem efficiently, or do not find exact answers, or do not assume the commonly-used word-wise machine model.

This paper proposes an indexing technique for computing range-temporal aggregates with guaranteed logarithmic access time. First, the plain range-temporal aggregate query is reduced into several subqueries. Next, a index

structure called the *Multiversion SB-Tree* (MVSB-tree) is proposed to solve these subqueries. The proposed structure incorporates features from both the *SB-Tree* [Yang and Widom 2001] and the *Multiversion B+-tree (MVBT)* [Becker et al. 1996]. This structure supports efficient queries and yet allows similarly efficient updates. In particular, computing a 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. Updates to the MVSB-tree are performed incrementally, as tuples are inserted. An update takes $O(\log_b K)$ I/Os, where K is the number of different keys inserted into the warehouse. Space is bounded by $O(\frac{n}{b} \log_b K)$.

Functional range-temporal aggregates can also be computed using the same structure (i.e., MVSB-tree) with constant augmentation on each index entry. Functional range-temporal queries are hence computed with the same worst-case guarantee on query and update performance as well as index size. This solution is extended to the case when a record has as value not a constant, but a general function.

The article substantially extends from the conference version [Zhang et al. 2001] of it, in the following ways.

(1) Extending the original problem to functional range-temporal aggregates.
(2) Presenting a more efficient query reduction technique for range-temporal aggregates.
(3) Providing formal proofs for lemmas and theorems.
(4) Covering recent related work.
(5) Introducing more intuitive examples.
(6) Expanding performance studies.

The major contributions of the article can be summarized as follows.

—It proposes the plain and functional range-temporal aggregate queries.
—It reduces the plain range-temporal aggregate query into dominance-sum queries. In particular, two versions of reduction are discussed. The preliminary version of the article [Zhang et al. 2001] reduces a range-temporal aggregate query into six dominance-sum queries. And this paper proposes a better reduction technique which needs four dominance-sum evaluations.
—It proposes the MVSB-tree, to solve the dominance-sum query (and in turn to solve the range-temporal aggregate queries). The index has a guaranteed worst-case bound on the query performance, update performance, and index size. While the straightforward solution to the range temporal aggregate problem has linear query performance, this structure succeeds in logarithmic time.
—It solves the functional range-temporal aggregate 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 when a general value function is involved.

The rest of the article is organized as follows. Section 2 discusses background and previous work. Section 3 presents techniques to reduce the plain range temporal aggregate query to simpler problems. These (and thus the plain range-temporal aggregate query) are solved using an MVSB-tree index, introduced and analyzed in Section 4. Section 5 solves the functional range-temporal aggregate problem. Section 6 presents results from our experimental comparisons. Finally, Section 7 concludes the article and provides future research directions.

2. BACKGROUND

This section first describes previous research on temporal aggregation and the related problem of point aggregation, then reviews the SB-tree and MVBT from which our proposed MVSB-tree draws ideas.

2.1 Temporal Aggregation

In the literature the term "temporal aggregation" has been used to represent the computation of a transformed temporal relation storing all temporal aggregates. Here an aggregate corresponds to a COUNT, SUM, AVG, MIN, or MAX of the records alive at one time instant. In more detail: a maximal continuous interval on which the aggregate values remain the same is called a *constant interval*. The process of grouping tuples over the constant intervals and computing their aggregate values is called *temporal grouping*. Temporal aggregations denote the operations that partition the time-line into constant intervals and perform temporal grouping on the constant intervals.

Kline and Snodgrass [1995] proposed the *aggregation-tree*, a main-memory structure based on the *segment tree* [Preparata and Shamos 1985], to incrementally compute temporal aggregates. Since the aggregation-tree is not balanced, the worst-case construction cost of it is $O(n^2)$. Kline and Snodgrass [1995] also presented a variant of the *aggregation tree*, the *k-ordered tree*, which is based on the k-orderliness of the base table. Gendrano et al. [1999], Ye and Keane [1997] and Gao et al. [2004] introduced parallel extensions to the approach presented in Kline and Snodgrass [1995]. Kim et al. [1999] proposed the *PA-tree* which reduces the construction cost to $O(n \log n)$. Similarly, Moon et al. [2000] used a balanced tree that modeled after red-black trees. These methods all reside in main memory.

Yang and Widom [2001, 2003] proposed the *SB-tree*, an incremental and disk-based index structure, to address the temporal aggregation problem without a predicate on the key range. Since our proposed structure extends the SB-tree, the structure will be reviewed in more detail in a later sub-section.

Tao et al. [2004] addressed an approximate version of the range-temporal aggregate problem, with the goal to reduce the index size by allowing the query result to be imprecise, but with bounded error. The proposed technique used the original reduction technique we proposed [Zhang et al. 2001], which reduces one range-temporal aggregate query into six queries. Although essentially all the six queries are dominance-sums, our original reduction divides the six queries into two categories: the LKLT query and the LKST query. Both types of queries take as input a time instant t and a key k. The LKLT query (which stands

for less-key & less-time) asks for the total value of objects whose keys are less than k and whose time intervals are strictly earlier than t. The LKST query (which stands for less-key & single-time) asks for the total value of objects whose keys are less than k and whose time intervals contain t. The proposed solutions in Tao et al. [2004] focus on answering the LKLT query and the LKST query approximately. There were two solutions proposed. The first solution utilizes the MVBT to achieve logarithmic query cost in the worst case. The second solution uses the off-the-shelf B+-tree and R-tree to achieve the same query cost in the expected case. Not surprisingly, by allowing the query result to be imprecise, both solutions in Tao et al. [2004] have much better index size. Another approximate temporal aggregation work is Tao and Xiao [2008], which solves the range-snapshot counting problem for a set of temporal records: given a time instant t and a key range R, find the number of records which are alive at t and whose keys are in R. A highly precise approximate solution is proposed. The error has a theoretical bound, and is typically less than 5% as revealed by experimental results. The solution uses linear space and has logarithmic query time and update time. This article focuses on computing range-temporal aggregates *precisely*.

Kang et al. [2004] proposed the *ITA-tree* for computing range-temporal aggregates. Unlike the transaction-time model assumed in this article (where updates happen in increasing time order), Kang et al. [2004] allow updates to happen in any order. Also, a query can contain multiple range predicates, one for each attribute. This solution, however, does not provide any guarantee on worst-case query performance. The ITA-tree is basically a B+-tree which indexes records on their start time instants. Thus unlike the SB-tree, there is no guarantee that at each level, a constant number of sub-trees should be recursively examined. The worst-case query performance is linear to the number of records. This article aims for very fast (logarithmic) query support.

2.2 Aggregate Computation over Rectangular Point

The reduction technique presented in Section 3 reduces the range-temporal aggregate query to the dominance-sum query, a special case of the rectangular point aggregate computation in that the query rectangle always uses the lower-left corner of space as its lower-left corner. This reduction essentially makes all the existing results on the point aggregate problem applicable to the former. This section surveys existing work on point aggregation.

Govindarajan et al. [2003] proposed the *Compressed Range B-tree* (CRB-tree), which is an external version of the *Compressed Range tree* [Chazelle 1988]. The CRB-tree uses $O(n/B)$ disk blocks and answers two-dimensional range-count queries in $O(\log_B n)$ I/Os. Here n is the number of points and B is the page capacity in number of entries. The solution was extended to handle range-sum queries. The basic idea is to store the weights along with the secondary structure. This solution achieves near linear space cost and $O(log_B n)$ I/Os query cost. However, in order to achieve such good space and query cost, Govindarajan et al. [2003] assumed a *bit wise* machine model. That is, any integer v is represented by exactly $\log_2 v$ bits. For instance, eight small integers,

each of which can be represented by 4 bits, can be squeezed into four bytes. As a comparison, the typical word-wise model uses four bytes to store a single integer. In general, implementing a structure with bit wise machine model is much harder than implementing a structure with wordwise model. So the CRB-tree is mainly of theoretical value, as no common computer architecture supports a bit-wise model. The model can be implemented on-top of standard architectures; however at the cost of an immense overhead. Its performance, furthermore, deteriorates with updates. This paper assumes the wordwise model.

Zhang et al. [2002] addressed the aggregate problem over multiple dimensions, assuming the typical word-wise model. They focus on the dominance-sum problem (find the total weight of objects to the lower-left of a query point), since the rectangular point aggregate problem is easily reducible to the dominance sum query. Note that here we use terms in the 2D space for clarity, but the solutions apply to higher dimensions. For instance, in higher dimensions, "to the lower-left" should be explained as "has a smaller coordinate value in all dimensions." Zhang et al. [2002] presented two ways of extending the internal-memory and static ECDF-tree, and found that the two versions have interesting tradeoffs. One version has good query performance but poor index size and update cost, and the other version has good index size and update cost, but poor query cost.

The best suggested index in Zhang et al. [2002] is the BA-tree, which results from augmenting the k-d-B-tree. As in the k-d-B-tree, every node corresponds to a rectangular space (the root node corresponds to the whole space), and the space of an index node is partitioned to the spaces of its child nodes by alternating partitioning dimensions. The augmentation the BA-tree has over the k-d-B-tree is that each index entry stores an X-border, a Y-border, and a subtotal value. Here the X-border corresponds to the X coordinates of the points below the index entry. The Y-border corresponds to the Y coordinates of the points to the left of the index entry. And the subtotal is the aggregate value of points to the lower-left of the index entry. The meanings of the X-border and Y-border will be clearer after studying the examples below.

Figure 2 illustrates the query processing algorithm in the BA-tree. The goal is to find the dominance-sum of a given query point q, i.e. the total weight of points in the shadowed region in Figure 2(a). The region can be partitioned into four parts. Figure 2(b) shows the region to the left of the index entry F (which contains q). Note that the total weight of points in this region is independent to the X coordinates of these points. The BA-tree keeps, along with index entry F, a "Y-border," which is a 1D structure that keeps the aggregate information of the Y coordinates of the points to the left of F. Similarly, Figure 2(c) shows the region below F, where the total weight can be calculated by query F's X-border. Figure 2(d) shows the region to the lower-left of F. This total weight is a fixed number as long as q is inside F, and therefore is kept as the subtotal of F. Finally, Figure 2(e) shows the region inside F, whose total weight is calculated by examining the child node which is referenced by F. Collectively, these augmented information allow answering a dominance-sum query by examining a single node at each level of the tree, rendering a query cost of $O(\log_b^2 n)$. Here

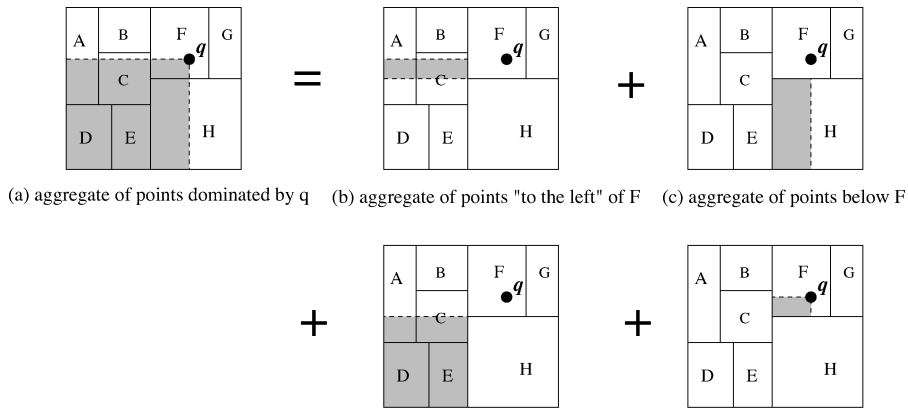


Fig. 2. Query processing in the BA-tree.

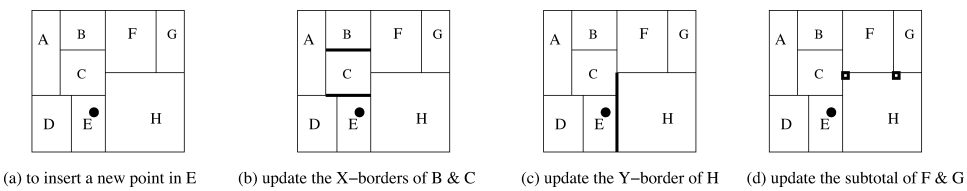


Fig. 3. Update in the BA-tree.

the additional $O(\log n)$ factor is needed for querying an X-border and a Y-border at each level of the tree.

The update process of the BA-tree is illustrated in Figure 3. To insert a new point into the index, like in the k-d-B-tree it is recursively inserted to the subtree whose spatial region contains the new point (referenced by E in Figure 3a). Besides, some X-borders, Y-borders, and subtotal values need to be updated. In particular, the X-borders of B and C needs to be updated (Figure 3b). To see the correctness, imagine a dominance-sum query is issued after the insertion, where the query point falls into B. It will be expected that the X value of the inserted point is recorded in the X-border of B. Similarly, the Y-border of H needs to be updated (Figure 3c). Finally, the subtotals of F and G needs to be updated (Figure 3d).

The only existing work we are aware of on functional aggregate computation is for spatial rectangular objects [Zhang et al. 2002]. There the addressed problem was to find the total value of objects intersecting a query rectangle, where the contribution of each object's value to the query result is proportional to the size of its intersection area with the query range. For example, in Figure 4, there are two objects intersecting the thick-bordered query region. The objects have values 4 and 3, which can be thought as how many grams of pesticide sprayed every square foot. These two objects intersect with the query region with sizes 50 and 12, respectively. Therefore the total volume of pesticide sprayed over the query region is $4 * 50 + 3 * 12 = 236$. The functional

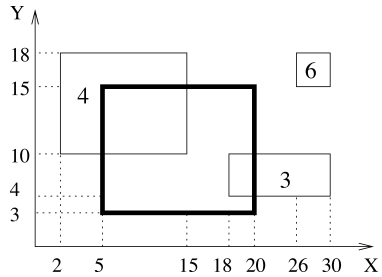
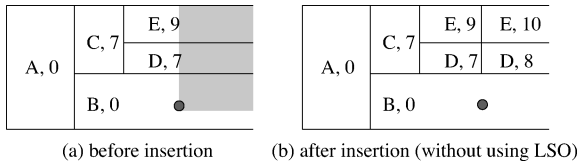


Fig. 4. Functional aggregate computation over spatial objects.



(a) before insertion (b) after insertion (without using LSO)

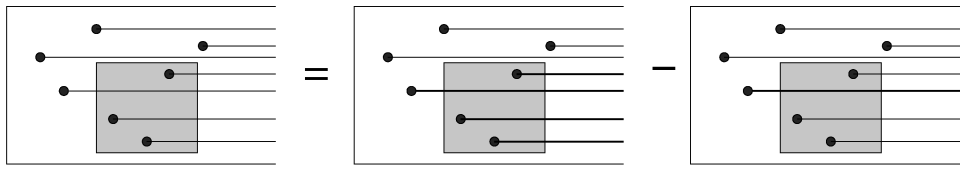
Fig. 5. Update in the MVSB-tree is much more efficient due to the transactional time model.

range-temporal aggregate query addressed in this paper is similar to that addressed in Zhang et al. [2002]. The difference is that in our problem the intersection area of every object (a line segment) with the query region is zero. In our problem, the contribution of each temporal object to the query result should be proportional to the time duration of the object inside the query region.

Even though the BA-tree, as a dominance-sum index, can be used to solve the range-temporal aggregate problem due to our reduction technique in Section 3, it is not efficient. The reason is that it does not utilize the special features of the temporal problem addressed in this article.

With the transaction-time model, the update shown in Figure 3 is not possible (assuming the X axis is time and the Y axis is key). The reason is that the new point has an update time earlier than the start time of some index entries, which violates the transaction-time model.

The new point in Figure 5 follows the transaction-time model, since it occurs at a time after all existing updates. This figure gives a precursor of an index node in the MVSB-tree (Section 4), which can solve the dominance-sum query in transaction-time model. The index needs to make sure that any dominance-sum query in the shadowed region in Figure 5(a) will take account the value of the newly inserted point (assume the value is 1). The point is recursively inserted into the subtree rooted by B, and therefore a later dominance-sum query is correct if the query point falls into the region of B. The problem is when the query point falls into E (or D). As shown in Figure 5(b), the MVSB-tree's idea is to break the index entry referencing E (or D) into two, and increase the aggregate value stored with the second copy of the index entry. This way, if a dominance-sum query point falls into the region of the index entry (E, 10), the dominance-sum result will include the value of the newly inserted point. Note that Section 4.2 will introduce the *logical splitting optimization (LSO)*, which makes sure that in an index node, a single entry needs to be updated. Compared



(a) the number of points in the rectangle (which is 3)

(b) the number of lines intersecting the right border (which is 4)

(c) the number of lines intersecting the left border (which is 1)

Fig. 6. The aP-tree converts points into lines and converts a point aggregate query into two 1D aggregate queries.

with the BA-tree where an update needs to update multiple X-borders and Y-borders and subtotal values, the MVSB-tree is much more efficient. Indeed, both query and update costs of the MVSB-tree have a factor of $O(\log_B n)$ speedup over the BA-tree approach.

Another point aggregation index is the aP-tree proposed by Tao and Papadias [2004]. The paper has a very interesting conversion. As illustrated in Figure 6, each 2D spatial point is converted to a horizontal line with right end open. A point aggregate query is then converted to two 1D aggregate queries: find the aggregate value of the points whose lines intersect the right border (or left border) of the query rectangle. This is a 1D aggregation query in that, if we consider the Y coordinates of all horizontal lines intersecting any particular X coordinate, we get a set of 1D values, and the query is to find, given a 1D range, the aggregate of objects in the range. This query can be answered in logarithmic time by an aB+-tree, which is a B+-tree where each index entry stores the aggregate of all objects in the sub-tree. To relax the restriction of considering a fixed X coordinate, one can make the aB+-tree partially persistent. This is exactly the aP-tree. It is similar to the MVBT. While the MVBT is a partially persistent B+-tree, the aP-tree is a partially persistent aB+-tree.

Although the aP-tree, as a point aggregation index, can also be used to solve the range-temporal aggregate query, it is an index that aims to store all the original objects. In the range-temporal aggregate problem this paper considers, multiple points to be inserted will have the same key. For instance, at the start and end times of the same phone call record, or at start or end times of multiple phone call records with the same phone number. In this case, our proposed MVSB-tree index is better because as a specialized aggregation index which does not care to store the inserted points themselves, multiple updates with the same key may be aggregated together. While the MVSB-tree has an update cost of $O(\log_b K)$ where K is the number of distinct keys, the aP-tree has an update cost of $O(\log_b n)$ where n is the number of inserted points.

2.3 The SB-tree

Since our proposed index structure, the MVSB-tree, draws ideas from the SB-tree [Yang and Widom 2001, 2003] we review the SB-tree in greater detail.

The SB-tree was proposed to answer the temporal aggregate queries without range predicates. In particular, given a set of temporal records, each having a time interval and a value, the *instantaneous temporal aggregate query* is to find

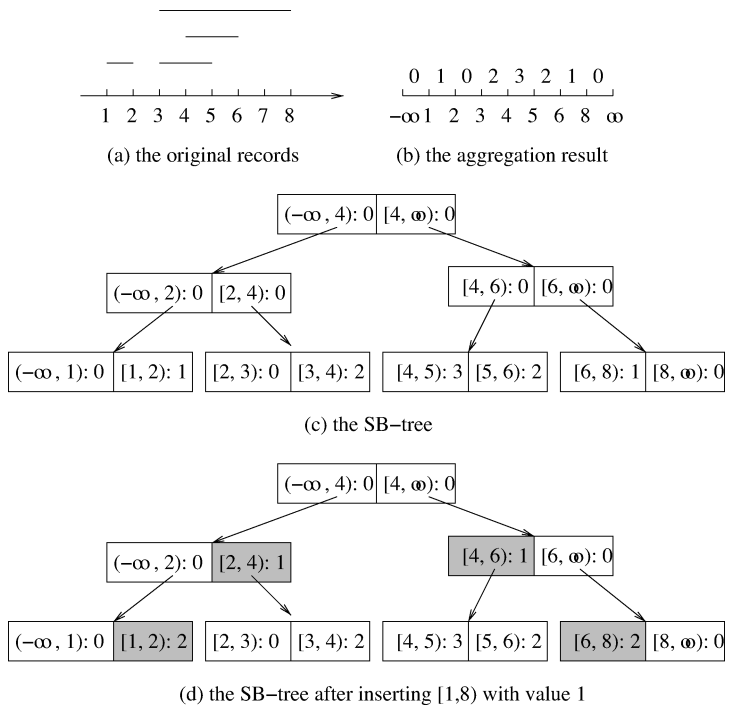


Fig. 7. Illustration of the SB-tree.

the total value of records whose intervals contain some time instant t, and the *cumulative temporal aggregate query* is to find the total value of records whose intervals intersect some time interval I.

The SB-tree incorporates properties from both the *segment tree* [Preparata and Shamos 1985] and the *B-tree*. The segment tree's features ensure efficient updates, even for tuples with long lifespans. 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, representing contiguous time intervals. Intervals are kept in both interior and leaf nodes. The interval associated with a node contains all intervals in the node's subtrees. For each interval, a value is associated with the record, storing the aggregate over this interval.

As an example, consider the four temporal records in Figure 7(a). Here for simplicity assume all objects have value = 1. An example of the instantaneous temporal aggregate query is to find the total value of, which in this case is the number of, objects whose intervals contain the time instant $t = 3.5$. Obviously, the query result should be 2. The SB-tree logically maintains the aggregate result for all time instants. Such aggregation result is shown in Figure 7(b). For instance, the value associated with the time interval [3,4) is 2. This means that the instantaneous temporal aggregate for any time instants $\in$[3,4) is 2.

One may wonder: what if a new record with time interval [1,8) is inserted into the base table? A naïve approach is: in Figure 7(b), update the aggregates

stored along with most of the intervals, except (-∞,1) and [8,∞). This approach has linear update cost. The SB-tree achieves logarithmic update performance by maintaining a paginated and balanced tree structure. As illustrated in Figure 7(c), the intervals keeping aggregate results are kept as leaf entries. Adjacent intervals are kept in a leaf node. As in the B-tree, every node except the root has between $b/2$ and b entries. Each leaf node is referenced by an index entry, which also has an interval (the union of all intervals in the leaf node). The index entries are kept in index nodes. Higher index levels can be recursively built. Each index entry keeps an aggregate value: the total number of intervals inserted, which fully covers the duration of the index entry. Note that Figure 7(c), where all index entries have aggregate value = 0, is for illustration purpose only. It is built from Figure 7(b), not by dynamically inserting the objects in Figure 7(a) into an initially empty SB-tree. But the tree is correct, in the sense that any instantaneous temporal aggregate query will get the correct result, which is stored with the corresponding leaf entry.

The key to reducing the update cost from linear time to logarithmic time is to make sure that the update process examines at most two index nodes at each level. These are the nodes referenced by the two index entries (at the parent level) whose intervals contain the two end points of I. Here I denotes the interval to be inserted. For an index entry whose interval is contained in I, we update the value stored along with the index entry, without going into the subtree and update the values of all leaf entries. For instance, Figure 7(d) shows the SB-tree after inserting [1,8) into the SB-tree of Figure 7(c). The entries whose values are modified are shadowed.

An instantaneous temporal aggregate is computed by recursively searching the SB-tree (starting from the root) and accumulating aggregate values of visited nodes. This results in fast aggregate computation time, namely, $O(\log_b n)$.

Two SB-trees are used to support cumulative SUM, COUNT and AVG aggregates with arbitrary window offset w. One SB-tree maintains the aggregates of records valid at any given time; the other maintains the aggregates of records that are valid strict before any given time. To compute an aggregate, the approach first computes the aggregate value $t + w$. It then adds the aggregate value of all records with intervals strictly before $t + w$ and finally subtracts the aggregate value of all records with intervals strictly before t.

2.4 Partially Persistent B-trees

In a *persistent* data structure, every update creates a new version of the data structure, while previous versions are still retained and can be accessed. In an *ephemeral* structure, in contrast, any old version is discarded. *Partial persistence* implies that updates are applied only to the latest version of the data structure, creating a linear ordering of versions. Clearly, partial persistence fits with the notion of transaction-time, version numbers replaced by the ordered sequence of time instants. As we will show, the MVSB-tree is an SB-tree that has been made partially persistent. The approach has been influenced by the MVBT [Becker et al. 1996] which is a structure that makes a B+ tree partially persistent.

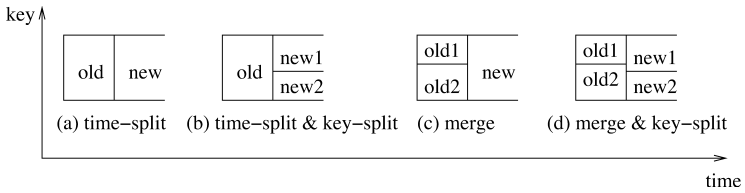


Fig. 8. Structural changes in an MVBT.

The MVBT keeps track of a set of temporal records, each having a key and a time interval. It 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. Here n is the number of records, and b is the page capacity.

The MVBT is a graph structure that maintains the evolution of a B+-tree over time. It has many roots, each responsible for the subsequent part valid during a specific time interval. References to the different roots associated with the corresponding time intervals are kept in an additional data structure called $root^*$. The MVBT partitions the key-time space into rectangles where each rectangle is associated with exactly one data page. A data record is stored in all the data pages whose key-time rectangle contains the data record's key and intersects the record's interval. The page rectangles are created recursively. As records are inserted into a certain page of a MVBT, the page may overflow. At that time, this page's currently alive data 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 (Figure 8a). 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 (Figure 8b).

Data records are inserted in the MVBT in increasing time order (i.e., *transaction-time* is assumed [Jensen and Snodgrass 1999]). When a data record is inserted at t, its deletion time is yet unknown and its interval is initiated to $[t, now)$; now is a variable representing the ever increasing current time. For implementation purposes, now is stored as $maxtime$. When later (if ever) this data record is deleted or updated, the end time in its interval is updated from now to the deletion time.

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 t-alive records, where d is linear to the page capacity. To achieve this, the MVBT uses yet another structural change: merge. If a *weak underflow* occurs after a deletion, that is, the key density of the page where the deletion takes place drops below the threshold d, the alive records in the page and a sibling page are copied into a new page (Figure 8c). To avoid frequent merge/splits, the MVBT requires that when a new page is created, the number of records in it must be between a lower bound and a higher bound (*strong condition*). If the result page of a merge operation has too many records (more than the upper bound), a key split is performed immediately (Figure 8d).

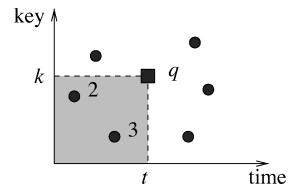


Fig. 9. A dominance-sum query.

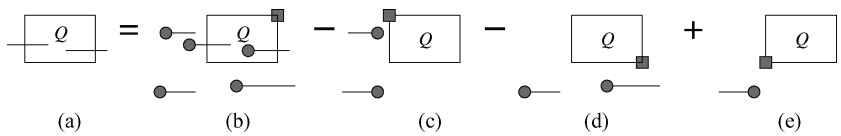


Fig. 10. Plain range-temporal aggregate query, reduced to 4 dominance-sum queries.

3. PROBLEM REDUCTION FOR THE PLAIN RANGE TEMPORAL AGGREGATE QUERY

In the preliminary version of this article [Zhang et al. 2001] we proposed a technique which reduces a plain range temporal aggregate query to six subqueries. In Section 3.1 we propose a new and better technique that reduces a query to four subqueries. The old reduction technique is described in Section 3.2. To get an overall picture, Section 3.3 provides the indexing scheme, update and query algorithms, with the assumption that a *dominance-sum index* exists. Section 4 proposes the MVSB-tree, solving dominance-sum queries efficiently.

3.1 Four Query Reduction

This section first defines dominance-sum queries and then shows how to reduce a plain range-temporal aggregate query to four such queries.

Definition 2. Let P be a set of point objects in the 2D time-key space. Each record consists of 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.

In Figure 9 the query point q dominates two objects (in the shadowed region) with values 2 and 3, respectively. Therefore the dominance sum for q is $2 + 3 = 5$.

THEOREM 1. *A plain range-temporal aggregate query can be reduced to four dominance-sum queries.*

PROOF. Intuitively, a query time interval and a key range collectively form a query rectangle Q in 2-dimensional time-key space. The plain range-temporal aggregate query asks for the total value of objects intersecting Q. The query rectangle Q has 4 corners. A range-temporal aggregate query is then reduced to 4 dominance-sum queries, one for each corner of the query box, as illustrated in Figure 10. In particular, Figure 10(a) shows a query rectangle intersected by two records. The range-temporal aggregate 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 10(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 10(c)). The false positives under Q correspond to those whose left corners are dominated by the lower right corner of Q (Figure 10(d)). After these false positives are subtracted from the query result, objects whose right corners are dominated by the lower left corner of Q (Figure 10(e)) have been subtracted twice. Hence, 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 compute range-temporal aggregates 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 aggregate query is reduced to two dominance-sum queries on the left corners and two dominance-sum queries on the right corners. □

3.2 Six Query Reduction

Our old reduction technique [Zhang et al. 2001] reduces a plain range-temporal aggregate query to two LKST queries and four LKLT queries. This section first describes these query types, then the reduction technique, and finally links the LKST and LKLT queries to 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 in S whose keys are less than k and whose time intervals contain t.

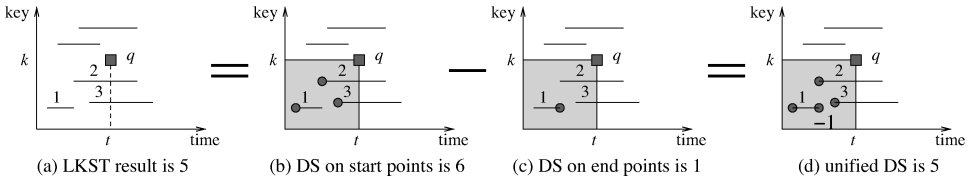
—The *less-key, less-time (LKLT) query* computes the total value of records in S whose keys are less than k and whose time intervals are strictly before t (i.e., whose *end* times are on or smaller than t).

THEOREM 2. *The plain range-temporal aggregate 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 total value of those tuples alive at t_2, plus the total value of those tuples alive strictly before t_2, minus the total value of those tuples alive strictly before t_1. This can be described by the following equation:

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

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



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

Fig. 11. The LKST query is also reduced to the dominance-sum query.

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:

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

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

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

To see that the LKST query can also be reduced to the dominance-sum query, consider Figure 11. In Figure 11(a), the LKST result, corresponding to $q = (t, k)$, is $2 + 3 = 5$. There are two records with values 2 and 3 whose time intervals contain t and whose keys are smaller than k. Notice that the start points of both records are dominated by q. But the dominance-sum of q on the starting points of the records (Figure 11(b)) is more than what is needed. The difference is the dominance-sum of q on the end points of the records (Figure 11(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 11(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 approaches in terms of indexing, insertion, and query schemes supporting the plain range temporal aggregate query, utilizing the MVSB-tree as the basic building block.

The *Four Query* approach is illustrated in Figure 12. It uses two MVSB-trees, one corresponding to the start points of the original temporal records and

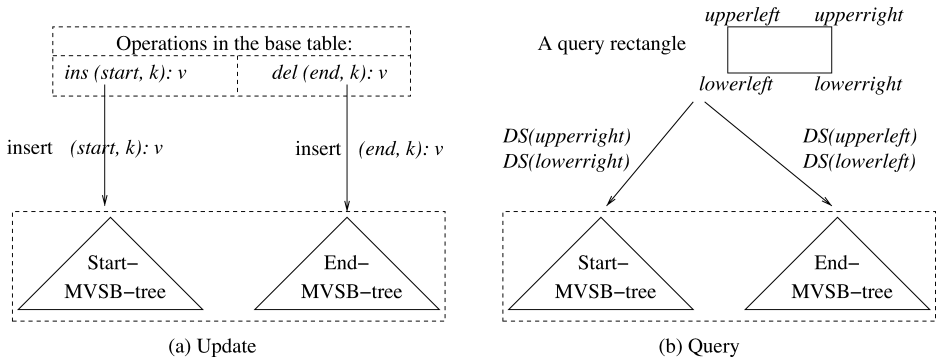


Fig. 12. The *Four Query* approach.

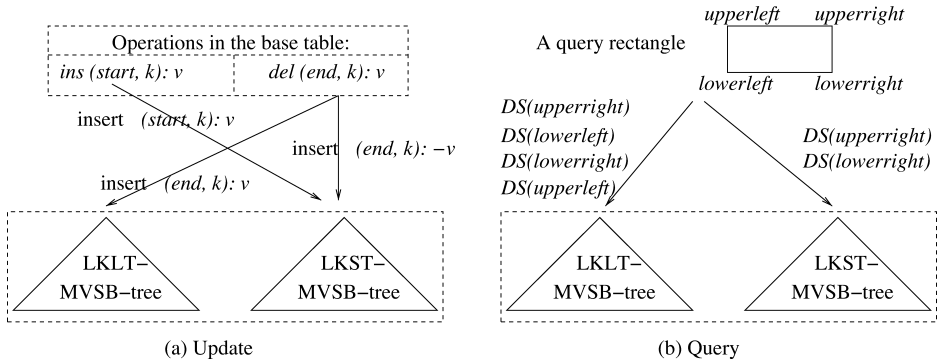
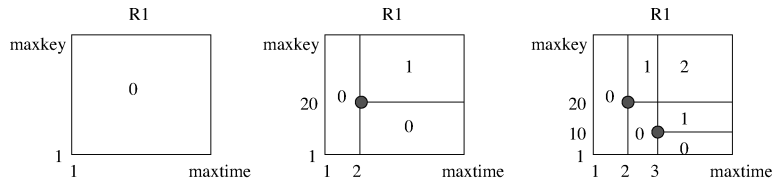


Fig. 13. The *Six Query* approach.

the other corresponding to end points. For ease of presentation we name the two trees *Start-MVSB-tree* and *End-MVSB-tree*. A single temporal record with $key = k$, $timeinterval = [start, end)$, and $value = v$, requires two updates, one in each MVSB-tree. The record's start point is inserted into the Start-MVSB-tree, and its end point is inserted into the End-MVSB-tree. To answer one plain range-temporal aggregate 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 10, the aggregate result is calculated as:

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

The *Six Query* approach is illustrated in Figure 13. It uses two MVSB-trees, one corresponding to the LKLT query and the other corresponding to the LKST query. We name the two trees *LKLT-MVSB-tree* and *LKST-MVSB-tree*. A single temporal record with $key = k$, $timeinterval = [start, end)$, and $value = v$, requires three updates. 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 aggregate query, six dominance-sum (DS) queries are performed, two in the LKLT-MVSB-tree and four in the



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

Fig. 14. A MVSB-tree after one and two insertions.

LKST-MVSB-tree. As Equation (1) shows, the aggregate result is:

$$DS_{LKST}(upperright) + DS_{LKLT}(upperright) \quad + DS_{LKLT}(lowerleft)$$

$$-DS_{LKST}(lowerright) - DS_{LKLT}(lowerright) - DS_{LKLT}(upperleft).$$

4. THE MULTIVERSION SB-TREE

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

—An update is to insert a 2D (time and key) point (t, k) with value v into the index. According to the transactional time model, objects are inserted in nondecreasing time order.

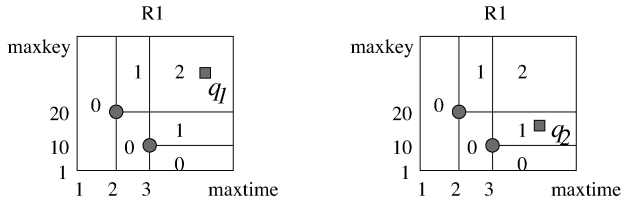
—A query is to find the total value of objects, ever inserted into the index, which are dominated by some query point in the time-key space.

4.1 The Initial MVSB-Tree with A Single Disk Page

Figure 14(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 which is 0. This value is the dominance-sum query result for any query point that falls inside of the rectangle of the record. To answer a dominance-sum query, the value of the record whose rectangle contains the query point is returned. Since no object has been inserted yet, any dominance-sum query will return 0.

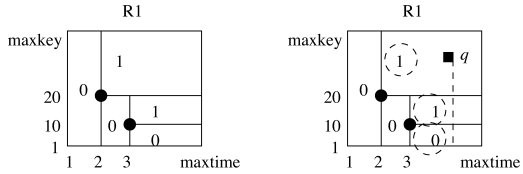
Suppose an object $\langle 2, 20\rangle : 1$ has been inserted, Figure 14(b). It is a point (2, 20) with value 1. The dominance-sum query result is 1 if the query point is located to the upper right of (2, 20), and 0 otherwise. The MVSB-tree implements this by splitting the original record into three. The rectangles of these three records do not overlap, and their union covers the whole space. The value of each record remains the dominance-sum query result for query points that fall inside the record's rectangle. To continue the example, Figure 14(c) shows the layout of the MVSB-tree after inserting $\langle 3, 10\rangle : 1$.

To see how dominance-sum query results are stored in the MVSB-tree, examine Figure 15(a). Query point q_1 dominates both inserted objects. The dominance-sum query result is the total value of these two objects, which is 2.



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

Fig. 15. Dominance-sum queries on a MVSB-tree.



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

Fig. 16. The counter part to Figure 14(c) using LSO.

Indeed the record whose rectangle contains q_1 has value 2. Figure 15(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.

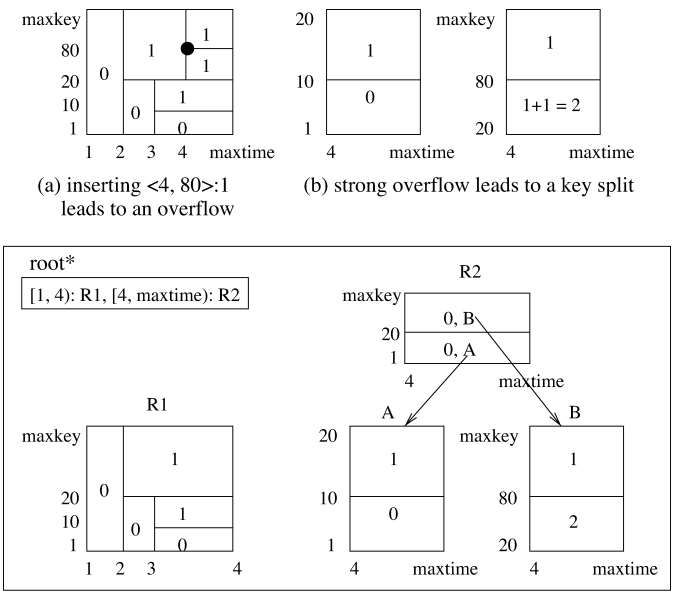
4.2 The Logical Splitting Optimization

In Figure 14(b), the right border of the rectangle (which corresponds to *maxtime*) has two segments: one with key range [1, 20), the other with key range [20, *maxkey*). Each segment corresponds to a different record stored in the page. A new insertion may split both records as shown in Figure 14(c). In general, one insertion may split many such records, incurring an enormous space cost.

We 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 14(c) with LSO is shown in Figure 16(a). In order to produce the correct dominance-sum query result, the query result is produced by aggregating the values of records below the query point. This is illustrated in Figure 16(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 alive records (whose rectangles end at *maxtime*) are copied to a new page. This is called a *time split*. However, the newly generated page may already be almost full. In such a case, a few subsequent insertions in the page



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

Fig. 17. Inserting $\langle 4, 80\rangle : 1$ to Figure 16(a).

trigger another time split, resulting in 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 in a page. 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) along the key dimension and the records are distributed evenly among these pages. Consider the insertion of $\langle 4, 80\rangle : 1$ into Figure 16(a). Assume $b = 6$, and $f = 0.5$. The insertion causes the record whose rectangle contains (4, 80) to split (Figure 17(a)). Notice that 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; it overflows. The four records touching the right border of space are copied to a new page. The strong condition states that this page should have no more than $f \cdot b = 3$ records. A strong overflow occurs and a key split takes place, distributing the records evenly into two pages (Figure 17(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 is adjusted by adding the total value of records in the other page. An index page is then 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 17(c).

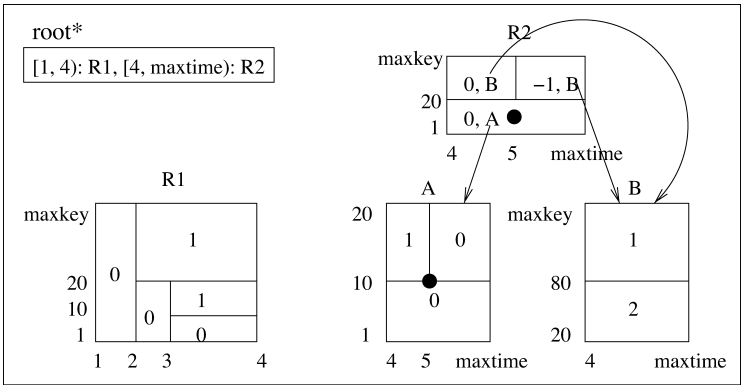


Fig. 18. Inserting $\langle 5, 10\rangle : -1$ to Figure 17(c).

4.3.2 *Updating an Index Entry.* An additional insertion introduces the update of index entries. Consider the insertion of $\langle 5, 10\rangle : -1$ into Figure 17(c). The insertion starts at 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. 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 B, we update the index entry pointing to page B. The result of the update is shown in Figure 18.

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 $o \in S$, $o.key \leq q$ iff $q \in$ [0,maxkey). Hence, the 1-dim dominance-sum problem can be transformed to:

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 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 reduce this 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 [Becker et al. 1996]. 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**, commonly 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 pointer 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 nonintersecting rectangles whose union equals the page's rectangle.*

For instance, in root R_1 of Figure 18, the five entries fully partition the space of R_1.

Since we assume that insertions come in nondecreasing 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, as discussed before all apply to the index page. These concepts become clearer after studying insertions.

Algorithm *DominanceSumQuery*
Input: An MVSB-tree *mvsbt*, Key k, Time t.
Output: the total value of inserted point objects whose keys $< k$ and whose times $< t$.

(1) *Let page be the root node in mvsbt whose life span contains t.*
(2) $v = 0$;
(3) ***while*** *page is an index page*
  (a) *Let rec be the index entry in page whose rectangle contains (t, k).*
  (b) *Derive the value of rec and add to v.*
  (c) *Let page be the node referenced by rec.*
  end while
(4) *Let rec be the leaf entry in page whose rectangle contains (t, k).*
(5) *Derive the value of rec and add to v.*
(6) ***return*** v.

Fig. 19. The algorithm to compute dominance-sum in an MVSB-tree.

4.4 Detailed Algorithms

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

Figure 19 shows the query algorithm. The algorithm searches a single path from root to leaf in the MVSB-tree. The nodes in the path are those whose rectangles in the time-key space contain the query point (t, k). Since every root node in the MVSB-tree corresponds to the whole key space, to find the root node whose rectangle contains (t, k), only the time t is used (Step 1). Along the path, each entry corresponds to a value. The aggregate of these values is the query result. However, each such value needs to be derived (Steps 3b and 5) due to the logical splitting optimization.

As an example, consider a dominance-sum query with $(t = 10, k = 100)$ in Figure 18. This query point dominates all the four points inserted into the initially empty MVSB-tree, namely: $\langle 2, 20\rangle : 1$, $\langle 3, 10\rangle : 1$, $\langle 4, 80\rangle : 1$, and $\langle 5, 10\rangle$:-1. The dominance-sum query result should be $1 + 1 + 1 - 1 = 2$. Let's see how the DominanceSumQuery algorithm in Figure 19 computes this. In Figure 18, the search path contains R_2 and B. The entries in them, whose rectangles contain the query point, are labelled "-1, B" in R_2 and "1" in B, respectively. But due to LSO, the values -1 and 1 are not the actual values of these entries. The actual values should be derived by aggregating the values of all entries "below" each of these two entries. That is, the value of the entry "-1, B" is $-1 + 0 = -1$, and the value of the entry "1" is $1 + 2 = 3$. Finally, the aggregate value of these entries, $-1 + 3 = 2$, is the correct query result.

Figure 20 is the insertion algorithm of the MVSB-tree. The insertion examines a single path from root to leaf (Step 1). These are the nodes whose rectangles contain the new point. Note that the insertion may not go all the way to the leaf level. If at some index node along the insertion path, there is no partly covered record, the insertion does not go to the subtree. For instance, in Figure 18, if the next point to be inserted has key = 20, the index entry in node

Algorithm *Insert*
Input: An MVSB-tree *mvsbt*, Key k, Time t, Value v.
Action: Insert $\langle t, k\rangle : v$ into *mvsbt*.

(1) *Starting from the last root in mvsbt, find the path of nodes whose rectangles contain the new point (t, k).*
(2) ***for*** *each node n in the path from bottom up,*
  (a) ***if*** *n is the lowest node in the identified path*
    i. *If there is a partly covered record in n, split it. Otherwise, split the lowest fully covered record.*
    ii. *If n overflows, split.*
  (b) ***else***
    i. *Split the lowest fully covered record.*
    ii. *If the child page has split, accommodate the entry pointing to the new child.*
    iii. *If n overflows, split.*
    end if
  end for
(3) *If the root node was split, create a new root.*

Fig. 20. The algorithm to insert a point in an MVSB-tree.

R_2 labelled "−1,B" will be split into two, but no leaf node needs to be modified.

Once the insertion path is fixed, the insertion is performed in a bottom-up fashion (Step 2). Step 2(a) and 2(b) differ in whether the insertion takes place in the lowest node in the path. There are two differences in the two cases. The first difference is that in the lowest page the algorithm tries to split the partly covered record. For instance, suppose an insertion takes place in Figure 18 at time = 6 and key = 5. There is a partly covered record in both node R_2 (labelled "0,A") and node A (the lower entry labelled "0"). But they are treated differently. In R_2, the partly covered record is not split. Rather, the fully covered entry "−1,B" should be split. But in A, the partly covered record is split. The other difference is that a nonlowest node may receive an additional reference to a newly generated child node.

Finally, if the root node splits, a new root is created (Step 3).

4.5 Record Coalescence and Page Coalescence

In this section, we describe two additional optimizations.

4.5.1 *The Record Coalescence Optimization.* The *Record Coalescence Optimization (RCO)* 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 coalesced either horizontally (*time merge*) or vertically (*key merge*). A time merge takes 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 21(a)). A key merge takes place, if (a) $lrec_1.interval = lrec_2.interval$; (b) $lrec_1.high = lrec_2.low$; and (c) $lrec_2.\ value = 0$ (Figure 21(b)).

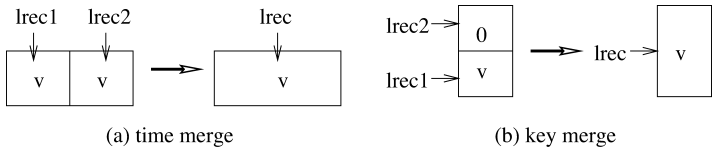


Fig. 21. Time and key merge.

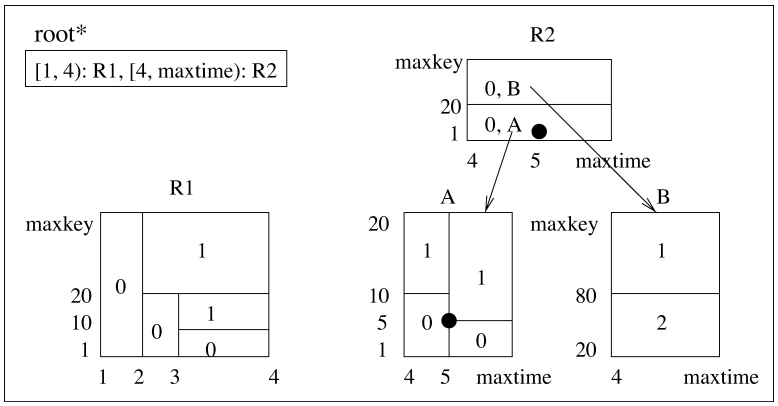


Fig. 22. Inserting $\langle 5, 5\rangle : 1$ to Figure 18.

Index records are coalesced similarly. The difference between merging index and leaf records is that two index records can be merged only if they point to the same child page.

Consider inserting $\langle 5, 5\rangle : 1$ in Figure 18. 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 22. One may wonder whether inserting a record at time $= 5$ violates the transactio-time model, since an insertion has already taken place at time $= 5$ and the transaction-time model prohibits updating history. The answer is that it still follows the slightly extended transaction-time model which allows multiple insertions at every version.

4.5.2 *The Page Coalescence Optimization.* Since we allow many insertions at the same time instant we only have to update the "net" effect of these insertions. However, our algorithms process one update at a time. The *Page Coalescence Optimization (PCO)*, 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 significant space.

4.6 Complexity Analysis

For ease of discussion, we do not consider record merging and the page disposal. Though these techniques improve performance, they are not required to meet

the worst-case bounds presented in the following. We first address the strong factor f. Due to the strong condition, there are at most $f \cdot b$ alive records in a newly created page. Note that although conceptually the MVSB-tree takes as input point objects, each inserted entity is really a right-side-open and top-side-open rectangle. That is why a page in the MVSB-tree has the notion of *alive* objects. To guarantee a fan-out of at least two, f has to be greater than $\frac{3}{b}$. Lemma 1 gives the maximal number of pages created due to the split of an overflowing page.

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

PROOF. The max number of alive records to be copied from the old page is $b+1$. Hence, 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 holds for all child pages of an index page p. If p overflows, the max number of alive records to be copied is $b + \lceil \frac{1.5}{f} + \frac{1}{3} \rceil - 1$. Hence, 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, an insertion in p can add of new records and logically deletion others. The number 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 arise 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 that 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; that is, $\forall i \in [1, x-1]$, 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 t_1 before $p_i.end$, there are at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at t_1. 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 t_1 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 with regards to K.

LEMMA 4. *The upper bound of the height of any subtree 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. Hence, 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. Hence, 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 and then traverses back, requiring a 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 covers the query time instant is known, it takes $O(\log_b K)$ I/Os to answer a point query in the worst case. If root* is kept as a B+-tree, extra I/Os are needed to locate the root. Since it takes at least $O(b)$ insertions for a root page to overflow (Lemma 2) after it has been generated, there are $O(n/b)$ root pages. Hence, 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 SB-trees embedded in the MVSB-tree (if a record is copied, the two copies are considered to occupy different slots). We show that each insertion creates $O(\log_b K)$ new occupied slots. The occupied slots are partitioned into two sets: in the first, occupied slots are created from copying existing occupied slots; all others slots belong to 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). Hence, 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. Hence, 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 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 Theorem 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 aggregate 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 aggregate 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 SB-tree roots ever created are kept in an array in main-memory. In this case, query time is reduced to traversing the appropriate SB-tree, i.e., $O(\log_b K)$.

5. FUNCTIONAL RANGE-TEMPORAL AGGREGATES

Section 5.1 reduces the functional range-temporal aggregate problem to a special case. Section 5.2 then proposes a solution to this special case, which in turn solves the general case.

5.1 Reduction to Origin-Involved Special Cases

To support functional range-temporal aggregate computation, 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*).

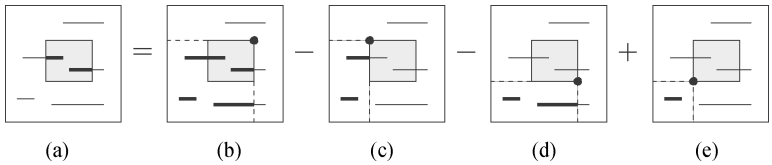


Fig. 23. Reduction from the functional range-temporal aggregate query to its origin-involved special case.

Figure 23(a) illustrates an arbitrary functional range-temporal aggregate 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 aggregate 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 23(b) illustrates another functional range-temporal aggregate 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 key-time space. We thus call such a query the *origin-involved special case*. Origin-involved queries only require specifying a single point in key-time space: the upper-right corner of the query rectangle. As shown in Figure 23, a functional range temporal aggregate query can be reduced to the origin-involved special case. To compute an arbitrary aggregate (Figure 23(a)), we first evaluate the origin-involved aggregate regarding the upper-right corner (Figure 23(b)). This query returns a larger value then desired, 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, we perform another origin-involved aggregate (Figure 23(c)). Similarly, we subtract those below the query rectangle (Figure 23(d)). The records both to the left and below the query rectangle (Figure 23(e)) have been subtracted twice, and thus need to be added back.

5.2 Origin-Involved Functional Range-Temporal Aggregates

We compute origin-involved functional aggregates as follows. Given a set of temporal records, for every point p in two-dimensional key-time space, the origin-involved functional aggregate regarding p is a single value. We design an index structure which logically maintains such values for all points in 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.

We first discuss the transformation of updates. Note that we adhere to the transaction time model, and updates to the original set thus only consist of insertions and deletions. For instance, the record in Figure 24(a) with $key = k_1$, $timeinterval = [t_1, t_2]$, and value v corresponds to two updates: an insertion

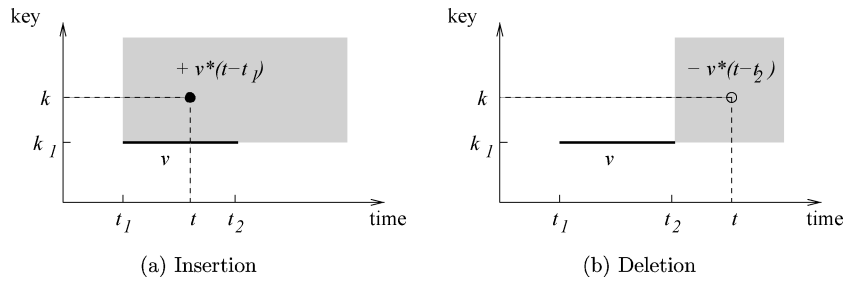


Fig. 24. Effect of the insertion/deletion of a temporal record.

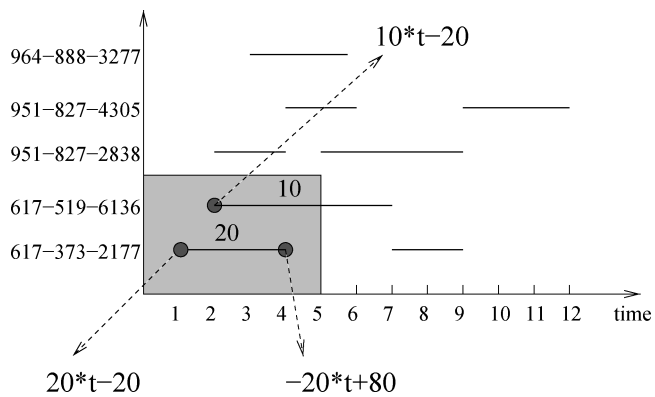


Fig. 25. An origin-involved functional range-temporal aggregate. The dominance-sum is $10 * t + 40$, e.g., 90 at $t = 5$.

at t_1 and a deletion at t_2. As Figure 24(a) shows, the insertion t_1, affects the hypothetical origin-involved aggregation index as follows: The value of each point $(k,\ t)$ in $[k_1,\ maxkey) \times [t_1,\ maxtime)$ is increased by

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

Similarly, as Figure 24(b) shows, deleting the temporal record at t_2, affects the index as follows: The value of each point $(k,\ t)$ in $[k_1, maxkey) \times [t_2,\ maxtime)$ is decreased by $v * (t - t_2)$. Or equivalently, increased by

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

Such updates have exactly the same format as the update of a MVSB-tree. The only difference is that in the MVSB-tree 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 use the MVSB-tree to compute the origin-involved functional temporal aggregates, with a slight modification that every aggregated "value" that is stored in the index is a pair of values instead of one.

Figure 25 shows the functions associated with three end points of objects, the points dominated by the upper-right corner of the shadowed query region.

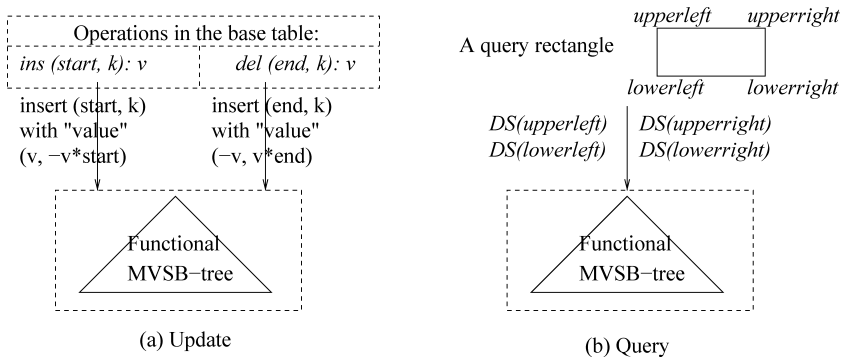


Fig. 26. Functional range-temporal aggregate.

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).

Such MVSB-trees allow evaluating functional range-temporal aggregates. 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 aggregate should be $10 * 3 + 20 * 3 = 90$. The MVSB-tree computes the dominance-sum as follows. 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

The insertion and query schemes supporting the functional range temporal aggregate query, utilizing the MVSB-tree as basic building block, are illustrated in Figure 26. In contrast to the plain range temporal aggregate case, only 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 23, a functional range-temporal aggregate is computed as:

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

Each dominance-sum $DS(\cdot)$ is first computed as a value function and then evaluated for the time of the query.

5.4 Extension to Support Value Functions

So far we discussed the case when the value for every record is a constant. In the functional aggregate 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.

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 $[k_1, maxkey) \times [t_1, maxtime)$

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 $[k_1,\ maxkey) \times [t_2,\ maxtime)$ 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 aggregate. 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 computed efficiently and 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 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 their coefficients), and can be evaluated efficiently. This means our solution for the continuous case of the functional aggregate works for polynomial temporal record functions, which is a rather general class of functions, covering many practical applications.

6. EXPERIMENTAL EVALUATION

This section evaluates the performance of the MVSB-tree based approaches for the plain and functional range-temporal aggregate query. The experimental setup is given in Section 6.1. Section 6.2 compares the query performance of MVSB-tree based approaches against the naïve approach of retrieving the records satisfying the range-interval condition and aggregating their values on the fly. The range-interval selection in the naïve approach is performed by querying a traditional temporal index, the MVBT [Becker et al. 1996]. Section 6.4 evaluates benefits of our three optimization techniques. Finally, Section 6.5 investigates the performance of functional temporal aggregates.

6.1 Experimental Setup

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

For 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 key-time space. The default value of R/I ratio is 1, and the default value of QRS is 1%.

The dataset in the experiments contains 1 million time intervals generated using the Time-IT software [Kline and Soo 1998]. Time space equals $[1, 10^8)$. 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 time intervals to these 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. After the temporal records were generated, the database was transformed such that each temporal record corresponds to two operations: an insertion and a (logical) deletion. These operations are applied in increasing time order to initially empty MVSB-trees.

6.2 Comparison with the Naïve Approach

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

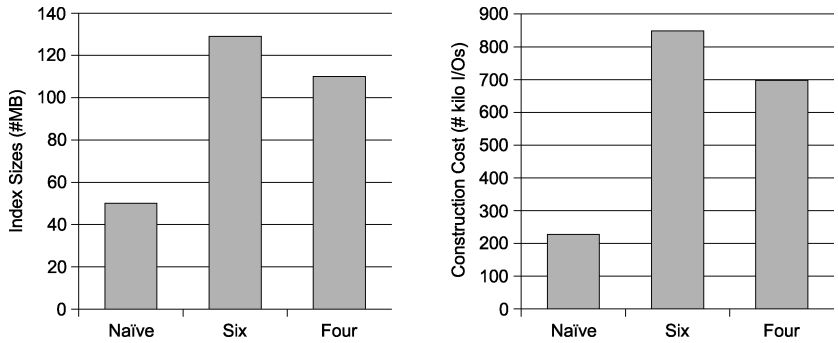
—*Naïve*: the naïve approach of using a range-interval selection query on an MVBT index to find the actual records and then aggregate their values on the fly.

—*Six*: the MVSB-tree-based approach using the old reduction technique, reducing a plain range-temporal aggregate query to **six** LKLT and/or LKST queries.

—*Four*: the MVSB-tree-based approach using the new reduction technique, which reduces a plain range-temporal aggregate query to **four** dominance-sum queries.

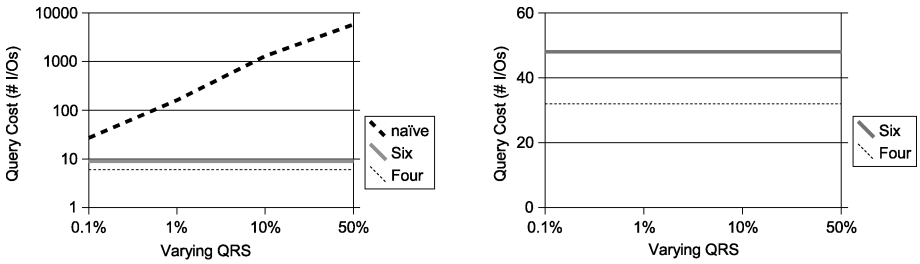
The goals of comparing these three algorithms are two-fold. First, we show how much faster our proposed approach (based on specialized aggregate index) is over the naïve approach (based on object retrievals). Second, compare the two versions of our new approach, using the six-query and four-query reduction techniques.

First, we compare the index size and generation cost of the three approaches. As Figure 27 shows, the MVSB-tree-based solution occupies more space and takes longer to generate. This is to be expected, since the both MVSB-tree



(a) Size (b) Cost

Fig. 27. Index size and construction cost.



(a) I/O cost (b) Elapsed time

Fig. 28. Query performance comparison, varying QRS.

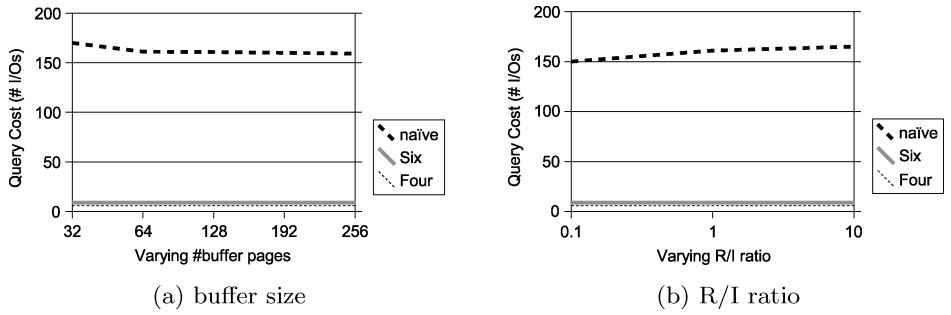
based approaches use two MVSB-trees, and each MVSB-tree has a $O(\log_b K)$ overhead in worst-case asymptotic space cost.

Comparing the two MVSB-tree-based solutions, *Four* is more efficient both in space cost and in construction cost. The reason is that *Four* performs two updates per time interval, while *Six* executes three.

Figure 28(a) shows that both versions of our approach are multiple orders of magnitude faster than the naïve approach. Note that the Y-axis is in logarithmic scale. The larger the QRS is, the more advantageous our approach becomes. This is to be expected, since the MVSB-tree-based query algorithms have logarithmic complexity, independent to the QRS, while the naïve approach has linear complexity.

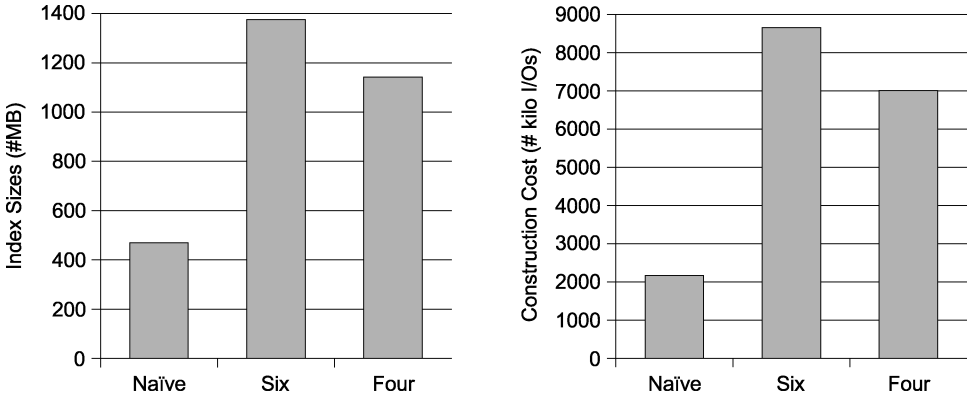
To better see the difference between *Six* and *Four*, Figure 28(b) focuses on comparing these two new approaches. Also, the figure displays the elapsed time (versus #I/Os) in regular scale (versus log scale). Between these two versions, *Four* is more efficient than *Six*. The reason is that *Four* answers a plain range-temporal aggregate query by four dominance-sum queries, and *Six* answers a plain range-temporal aggregate query by six dominance-sum queries.

Figure 29(a) and (b) compares the query performance of the three algorithms, for varying the buffer size and R/I ratio; QRS remains at the default value (1% of the key-time space). In both cases *Four* is slightly more efficient than *Six*,



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

Fig. 29. Query performance for varying buffer size and R/I ratio.



(a) Size (b) Cost

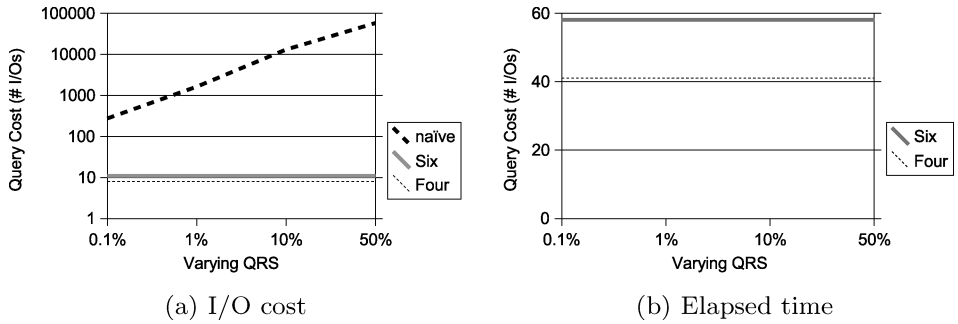
Fig. 30. Index size and construction cost using the 10-million-record dataset.

and both are clearly superior to the naïve approach. In the remaining sections, we focus on evaluating the *Four* algorithm alone.

6.3 Scale Up

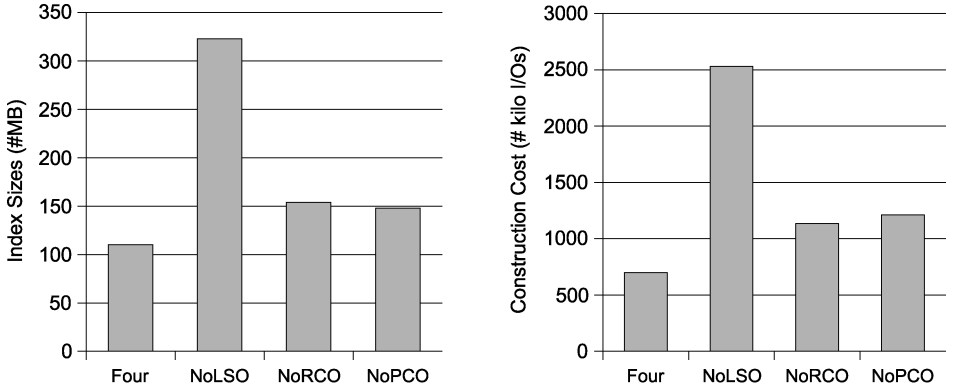
In order to study how the proposed schemes scale up with larger data sets, this section uses a larger dataset. Instead of 1 million records, this section uses 10 million records, again generated using the Time-IT software [Kline and Soo 1998]. There are 10,000 keys and each key corresponds to 1000 intervals.

Figure 30 compares the index size and construction cost. Figure 31 compares the query performance. Compared with their counterparts using the 1-million-record dataset (Figure 27 and Figure 28), the trends and conclusions are similar. The index sizes and construction cost are about 10 times larger. The query cost of the MVBT is about 10 times larger, too. An interesting observation is that the query cost of the MVSB-tree-based approaches is NOT 10 times larger! In fact the cost only slightly increased. The reason is that the MVSB-tree has a query cost logarithmic to the number of distinct keys, while the two datasets have the same number of distinct keys.



(a) I/O cost (b) Elapsed time

Fig. 31. Query performance comparison, varying QRS, using the 10-million-record dataset.



(a) Size (b) Cost

Fig. 32. Impact on index size and construction cost.

6.4 Effect of the Optimizations

This section evaluates the benefits of our three optimization techniques, namely:

—**LSO**: Logical Splitting.
—**RCO**: Record Coalescence.
—**PCO**: Page Coalescence.

Figure 32(a), Figure 32(b), and Figure 33 demonstrate the impact the optimization techniques have on the index size, construction cost, and the query performance. Here **Four** stands for the same algorithm we used in the previous section. That is, the MVSB-tree-based index solution for the plain range-temporal aggregate query, utilizing the four query reduction technique. **NoLSO** denotes *Four* without LSO, and **NoRCO** denotes *Four* without RCO. Finally **NoPCO** is *Four* without PCO.

Clearly LSO is the most important of the three optimization techniques. 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. The LSO optimization hence brings enormous

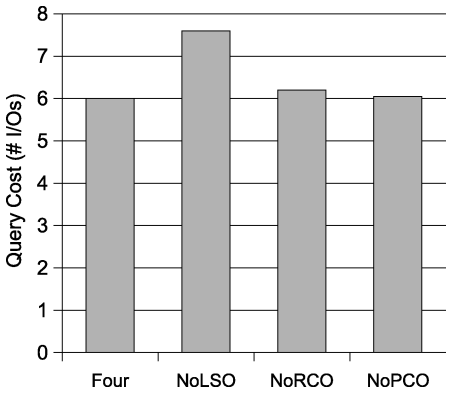


Fig. 33. Impact on query cost.

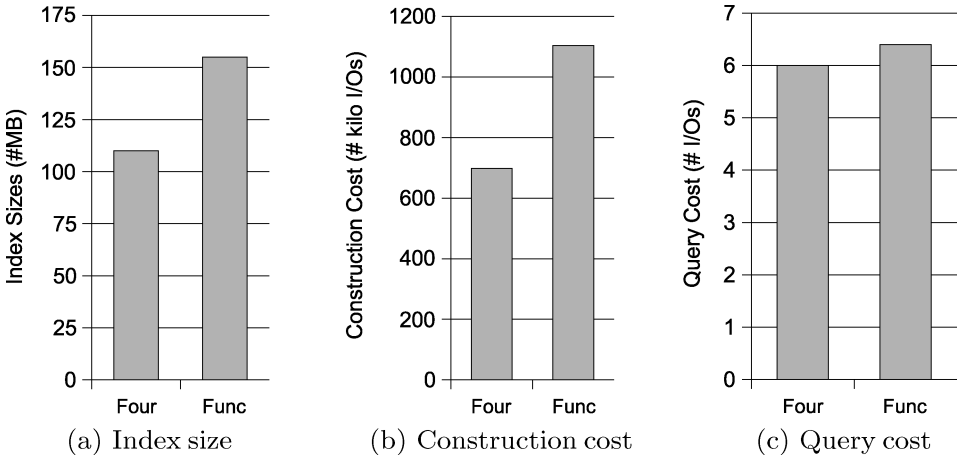


Fig. 34. Functional temporal aggregate, compared to using the plain temporal aggregate.

savings in index size and update cost. With a more compact index, the query cost of the solution with LSO is also noticeably smaller. The other two optimizations also show significant impact, but certainly not as big.

6.5 Functional Range Temporal Aggregate Query

Figure 34 shows performance results of the functional temporal aggregate solution. As baseline for comparison, we use the data of the plain temporal aggregate 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 to start times of records, the other corresponding to end times. A single record generates two updates, one in each MVSB-tree. 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. The second however suggests that the functional case may require less space. This is because maintaining an MVSB-tree index has some overhead, for example, the root structure, and the functional case saves on such overhead by maintaining one index instead of two. The net effect, as shown in Figure 34(a), is that the functional case uses little more space than the plain case. Consequently, update (Figure 34(a)) and query (Figure 34(b)) are also a little more expensive. In all cases, the functional solution is rather cheap, making it a reasonable extension of the plain case.

7. CONCLUSIONS AND FUTURE WORK

Temporal aggregates have become predominant operators in analyzing historical data. This paper examined temporal aggregate queries in the presence of key-range predicates. Such queries allow warehouse managers to focus on tuples grouped by some key range over a given time interval. We considered both plain and functional range-temporal aggregates. These problems are reduced to dominance-sum queries. We proposed the Multiversion SB-Tree (MVSB-tree) for incrementally maintaining and efficiently computing the dominance-sum queries and in turn range-temporal aggregate 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 are verified through an experimental evaluation.

One future direction is to try to prove the MVSB-tree is asymptotically optimal if exact answers are required. For large-scale applications where an index with nonlinear space is not acceptable, one should explore approximate solutions preferably allowing the users to control the trade-off between index size and query accuracy. Finally, it is interesting and challenging to relax the transaction-time model and propose efficient index solutions.

REFERENCES

BECKER, B., GSCHWIND, S., OHLER, T., SEEGER, B., AND WIDMAYER, P. 1996. An asymptotically optimal multiversion B-tree. *VLDB J. 5,* 4, 264–275.

CHAZELLE, B. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput. 17*, 427–462.

GAO, D., GENDRANO, J. A. G., MOON, B., SNODGRASS, R. T., PARK, M., HUANG, B. C., AND RODRIGUE, J. M. 2004. Main memory-based algorithms for efficient parallel aggregation for temporal databases. *Distrib. Paral. Datab. 16,* 2, 123–163.

GENDRANO, J., HUANG, B., RODRIGUE, J., MOON, B., AND SNODGRASS, R. 1999. Parallel algorithms for computing temporal aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*. 418–427.

GOVINDARAJAN, S., AGARWAL, P. K., AND ARGE, L. 2003. CRB-Tree: An efficient indexing scheme for

range-aggregate queries. In *Proceedings of International Conference on Database Theory (ICDT)*. 143–157.

JENSEN, C. S. AND SNODGRASS, R. 1999. Temporal data management. *IEEE Trans. Knowl. Data Engine. (TKDE) 11,* 1, 36–44.

KANG, S. T., CHUNG, Y. D., AND KIM, M.-H. 2004. An efficient method for temporal aggregation with range-condition attributes. *J. Inform. Sci. 168,* 1-4, 243–265.

KIM, J. S., KANG, S. T., AND KIM, M. H. 1999. Effective temporal aggregation using point-based trees. In *Proceedings of International Conference on Database & Expert Systems Applications (DEXA)*. 1018–1030.

KLINE, N. AND SNODGRASS, R. 1995. Computing temporal aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*. 222–231.

KLINE, N. AND SOO, M. 1998. Time-IT, the time-integrated testbed. ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz.

MOON, B., LOPEZ, I., AND IMMANUEL, V. 2000. Scalable algorithms for large temporal aggregation. In *Proceedings of International Conference on Data Engineering (ICDE)*. 145–154.

PREPARATA, F. AND SHAMOS, M. 1985. *Computational Geometry: An Introduction*. Springer Verlag.

SALZBERG, B. AND TSOTRAS, V. J. 1999. Comparison of access methods for time-evolving data. *ACM Comput. Surv. 31,* 2, 158–221.

TAO, Y. AND PAPADIAS, D. 2004. Range aggregate processing in spatial databases. In *IEEE Trans. Knowl. Data Engin.* 1555–1570.

TAO, Y., PAPADIAS, D., AND FALOUTSOS, C. 2004. Approximate temporal aggregation. In *Proceedings of International Conference on Data Engineering (ICDE)*. 190–201.

TAO, Y. AND XIAO, X. 2008. Efficient temporal counting with bounded error. *VLDB J.*.

YANG, J. AND WIDOM, J. 2001. Incremental computation and maintenance of temporal aggregates. In *Proceedings of International Conference on Data Engineering (ICDE)*. 51–60.

YANG, J. AND WIDOM, J. 2003. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) 12,* 3, 262–283.

YE, X. AND KEANE, J. 1997. Processing temporal aggregates in parallel. In *Proceedings of the International Conference on Systems, Man, and Cybernetics*. 1373–1378.

ZHANG, D., MARKOWETZ, A., TSOTRAS, V. J., GUNOPULOS, D., AND SEEGER, B. 2001. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the ACM International Symposium on Principles of Database Systems (PODS)*. 237–245.

ZHANG, D., TSOTRAS, V. J., AND GUNOPULOS, D. 2002. Efficient aggregation over objects with extent. In *Proceedings of the ACM International Symposium on Principles of Database Systems (PODS)*. 121–132.

Received April 2007; revised May 2007, October 2007; accepted December 2007