Efficient Computation of Temporal Aggregates with Range Predicates

Donghui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos,
Bernhard Seeger

December 11, 2000

TR-52

A TimeCenter Technical Report

Title	Efficient Computation of Temporal Aggregates with Range Predicates
Author(s)	Donghui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos, Bernhard Seeger
Publication History	December 2000. A TimeCenter Technical Report

Extended Abstract

A temporal aggregation query is an important but costly operation for applications that maintain time-evolving data (data warehouses, temporal databases, etc.). Due to the large volume of such data, performance improvements for temporal aggregation queries are critical. In this paper we examine techniques to compute temporal aggregates that include key-range predicates (*range temporal aggregates*). In particular we concentrate on SUM, COUNT and AVG aggregates. This problem is novel; to handle arbitrary key ranges, previous methods would need to keep a separate index for every possible key range. We propose an approach based on a new index structure called the *Multiversion SB-Tree*, which incorporates features from both the SB-Tree and the Multiversion B-Tree, to handle arbitrary key-range temporal SUM, COUNT and AVG queries. We analyze the performance of our approach and present experimental results that show its efficiency.

1 Introduction

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

Many approaches have been recently proposed to address temporal aggregation queries ([Tum92, KS95, YK97, GHR+99, MLI00, YW01]). However, all previous research has considered the so-called *scalar* temporal aggregates, where the temporal aggregates are computed over the *whole* key range of the database relation. The most efficient among the previous approaches is the SB-tree ([YW01]) which in addition can restrict temporal aggregation to a given time interval. Here we address the more general *range-temporal aggregation* problem (RTA). In this problem, the temporal aggregate is restricted by a time interval AND a *key range*. Since historical warehouses have large sizes, the RTA query is a very useful and practical operation as it enables the warehouse manager to focus the aggregation to any time-interval and/or key-range in the warehouse.

The paper concentrates on the SUM, COUNT and AVG aggregates. We first reduce the RTA query into two subproblems, namely the *less-key, single-time* query and the *less-key, less-time* query. We then propose a new index structure called the *Multiversion SB-Tree* (MVSBT) to solve these queries. The proposed structure incorporates features from both the *SB-Tree* ([YW01]) and the *Multiversion B-Tree (MVBT)* ([BGO+96]). By using two MVSBTs we can maintain and compute RTA queries very efficiently. In particular, computing an RTA takes $O(\log_b n)$ I/Os, where b is the capacity of a disk page and n is the number of tuples in the warehouse. Updating the MVSBT is done incrementally as tuples are updated (an update takes $O(\log_b K)$ I/Os, where K is the number of different keys inserted into the warehouse). The space is bounded by $O(\frac{n}{b} \log_b K)$.

We compare the performance of our approach against using a single index that first retrieves the tuples of the warehouse which satisfy the RTA key-range and time-interval predicates, and then computes the aggregate on the retrieved tuples. Possible choices for this index is a traditional multidimensional index (like an R*-tree [BKS+90]) or a temporal index (like the MVBT [BGO+96] or the TSB-tree [LS89]). We use the MVBT since it optimally solves a *range-snapshot* query ("find all tuples with keys in range r that were alive at time t"). Our initial experimental results show that our approach provides superior performance in computing RTA queries at the expense of a small space overhead. It should be noted that since the temporal aggregation query can involve arbitrary key ranges none of the previously proposed scalar methods is applicable. For example, the obvious approach of having a separate SB-Tree for each possible key range will not be efficient because of the large space requirements.

The rest of the paper is organized as follows. Section 2 discusses background and previous work. The problem reduction is addressed in section 3. The MVSBT index is introduced and analyzed in section 4. Section 5 discusses results from our experimental comparisons. Finally, section 6 presents conclusions and open problems for further research.

2 Background

We first describe previous research on temporal aggregation queries including the SB-tree. We then discuss the temporal data model assumed in our work and provide a short description of the MVBT.

2.1 Previous work on temporal aggregates

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

[Tum92] presents a non-incremental two-step approach where each step requires a full database scan. First the intervals of the aggregate result tuples are found and then each database tuple updates the values of all result tuples that it affects. This approach computes a temporal aggregate in $O(mn)$ time, where m is the number of result tuples (at worst, m is $O(n)$; but in practice it is usually much less than n). Note that this two-step approach can be used to compute range-temporal aggregates, however the full database scans makes it inefficient. [KS95] uses the aggregation-tree, a main-memory tree (based on the segment tree [PS85]) to incrementally compute temporal aggregates. However the structure can become unbalanced which implies $O(n)$ worst-case time for computing a scalar temporal aggregate. [KS95] also presents a variant of the aggregation tree, the k-ordered tree, which is based on the k-orderness of the base table; the worst case behavior though remains $O(n)$. [GHR+99, YK97] introduce parallel extensions to the approach presented in [KS95]. [MLI00] presents an improvement by considering a balanced tree (based on red-black trees). However, this method is still main-memory resident. A seminal work on incremental, disk-based, scalar temporal aggregate computation appears in [YW01], where the SB-tree index is introduced. Since our work draws from the SB-tree we will discuss its properties below; for more details we refer to [YW01].

2.2 The SB-tree

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

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

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

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

2.3 Temporal Data Model

For simplicity, we assume that each tuple in the warehouse is stored as a record that contains a *key*, a *time interval* and an attribute whose *value* is to be aggregated. We follow the *First Temporal Normal Form (1TNF)* ([SS88]) which specifies that there are no two tuples with equal keys and intersecting intervals. Without loss of generality, we assume that both keys and time instants are positive integers. Let the key space be $[1, maxkey)$ and the time space be $[1, maxtime)$. A time interval (or *interval* in short) has the form: $[start, end)$ where $1 \leq start < end \leq maxtime$. An interval reduces to a time instant when $end = start + 1$. Similarly, a key range (or *range* in short) has the form $[low, high)$ where $1 \leq low < high \leq maxkey$. A range reduces to a key when $high = low + 1$. For two ranges *r1, r2* which do not intersect, we say *r1* is *lower* than *r2* if $r1.high \leq r2.low$. A record *rec* is *alive* at time t if $t \in rec.interval$. A rectangle R in the key-time space consists of an key range *R.range* and a time interval *R.interval*. A record x is *in* rectangle R if $x.key \in R.range$ and *x.interval* intersects with *R.interval*.

When considering temporal data, it is important to distinguish the time model used by the temporal application. In the temporal database literature two time dimensions have been proposed, namely the valid-time and the transaction-time ([J+98]). The kind of updates supported on the temporal data depends on whether valid-time or transaction-time (or both) is supported ([KTF98]). In a valid-time environment when a tuple is inserted in the database, its associated interval is fully known. Moreover, tuples can be added and deleted from the database in any order. After a tuple is deleted its record is physically removed from the database (and thus cannot be further queried). The SB-tree has been designed for the valid-time environment. In contrast, a transaction-time environment assumes that tuple updates arrive in the database ordered by time. Hence, when a tuple is inserted at time t_i, its record's interval is initiated as $[t_i, now)$ where *now* is a variable representing the ever increasing current time (in practice, variable *now* is stored as *maxtime*). However, a tuple deletion is not physical but logical. For example, if the above tuple is deleted at time t_j its record's interval *end* is updated from *now* to t_j. That is, the record is still maintained in the database and can be queried. Since deletions are logical, in a transaction-time environment we cannot change the past. Equivalently, the transaction-time model maintains the history of a time-evolving database. The ability to change the past is useful in cases where errors are discovered in the recorded information.

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

2.4 Partially Persistent B-trees

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

Conceptually, the MVBT is a graph that maintains the evolution of a B+-tree over time. It has many roots, each responsible for accessing the B+-tree as it was during a specific time interval. The

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

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

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

3 Problem Reduction

Since AVG = SUM / COUNT, we focus on SUM and COUNT. Below we reduce an RTA query for SUM(COUNT) to two subqueries.

Definition 1 *Given a temporal relation T, key k and time t, a* ***less-key, single-time (LKST)*** *query finds the aggregate value of all tuples from T whose keys are less than k and whose intervals contain t.*

Definition 2 *Given a temporal relation T, key k and time t, a* ***less-key, less-time (LKLT) query*** *finds the aggregate value of all tuples from T whose keys are less than k and whose end times are less than or equal to t.*

Intuitively, a tuple with $end \leq t$ has been alive strictly before t.

Theorem 1 *Solving the RTA query for SUM and COUNT is reduced to solving the LKST and the LKLT queries.*

Proof. We use SUM in the proof since the same discussion holds for COUNT. Let the query rectangle be $[k_1, k_2)$ x $[t_1, t_2)$. Let $t_3 = t_2 - 1$ and let $r = [k_1, k_2)$. If we only consider tuples with keys in r, the SUM of the values of tuples whose intervals intersect $[t_1, t_3]$ is equal to the the SUM of the values of those tuples alive at t_3 plus that of those tuples alive strictly before t_3 minus that of those tuples alive strictly before t_1. This can be described by the following equation:

$$SUM(r, [t_1, t_3]) = SUM(r, t_3) + SUM(r, end \leq t_3) - SUM(r, end \leq t_1).$$

We now consider all the tuples alive at t_3. $SUM(r, t_3)$ can be computed as the SUM of the values of the tuples whose keys are less than k_2 minus the SUM of the tuples of the records whose keys are less than k_1. Or,

$$SUM(r, t_3) = SUM(key < k_2, t_3) - SUM(key < k_1, t_3) = LKST(k_2, t_3) - LKST(k_1, t_3)$$

Similarly, we have:

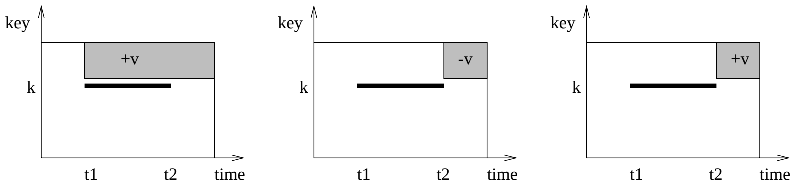
$$\begin{aligned} SUM(r, end \leq t_3) &= LKLT(k_2, t_3) - LKLT(k_1, t_3) \\ SUM(r, end \leq t_1) &= LKLT(k_2, t_1) - LKLT(k_1, t_1). \end{aligned}$$

Hence, we get:

$$\begin{aligned} RTA([k_1, k_2), [t_1, t_3]) &= LKST(k_2, t_3) + LKLT(k_2, t_3) + LKLT(k_1, t_1) \\ &- LKST(k_1, t_3) - LKLT(k_1, t_3) - LKLT(k_2, t_1) \end{aligned} \tag{1}$$

Thus a RTA query is reduced to two LKST queries and four LKLT queries. □

In order to support the RTA queries, we will present an access method that combines two index structures: one index supporting the LKST queries and the other supporting the LKLT queries. According to equation 1 above, an RTA aggregation query is transformed to six point queries for the LKST and LKLT indices. It remains to show how inserting and updating a temporal tuple is represented by each of the LKST and LKLT indices. Inserting a new tuple affects only the LKST index. In particular, to insert a tuple with key k and value v at time t_1, the LKST index should add v to all the points in $[k+1, \mathit{maxkey})$ x $[t_1, \mathit{maxtime})$ (figure 1a). We denote such an operation in the LKST index as an insertion of $\langle k+1, t_1\rangle : v$. Logically deleting a tuple affects both indices. To logically delete the above tuple at a later time t_2, the LKST index should subtract value v from, or equivalently, add value $-v$ to, all the points in $[k+1, \mathit{maxkey})$ x $[t_2, \mathit{maxtime})$ (figure 1b). This is denoted as an insertion of $\langle k+1, t_2\rangle : -v$. This logical deletion is also transformed into an insertion in the LKLT index: add value v to all the points in the rectangle $[k+1, \mathit{maxkey})$ x $[t_2, \mathit{maxtime})$ (figure 1c), which is denoted as $\langle k+1, t_2\rangle : v$.



(a) insertion:LKST (b) deletion:LKST (c) deletion:LKLT

Figure 1: Transforming the insertion and (logical) deletion operations

Hence both the LKST and LKLT indices can be implemented by the same structure. This structure should support: (1) Efficient insertion operations of the form: "given key k, time t and value v, add v to the values associated with all the points in the rectangle [k, *maxkey*) x [t, *maxtime*)"; (2) Efficient point queries as in: "given key k and time t, find the value associated with this point in the key-time space".

Now we focus on designing such a structure. First, let's assume the time dimension is fixed to some time instant t and focus on the key dimension. The structure should logically store a value at every key in the key space. This is expensive, since there are many keys. If adjacent keys store the same value, the keys can be combined into a key range. So what really should be stored is a set of non-intersecting key ranges whose union is the key space, where a value is associated with each range. Now, the insertion operation needs to update all the stored ranges that intersect [k, *maxkey*). Obviously, the smaller k is, the more ranges need to be updated. Our goal is to have an structure whose insertion time is independent to where k is. This scenario reminds of the SB-tree which supports efficient insertions of a time interval independently to where and how long the time interval is. So, by using an SB-tree for the key dimension, the requirement of efficient insertion is satisfied. The requirement of efficient point query is also satisfied, since the SB-tree is efficient in finding the value associated with any given point.

So far we have found the solution for a fixed time instant. To find a solution for the whole time space, a natural extension is to make an SB-tree partially persistent. Logically, the partially-persistent SB-tree (also called Multiversion SB-tree) is equivalent to a series of SB-trees, one at each time instant. An insertion operation and a point query involving time t are directed to the SB-tree corresponding to t. Physically, of course, it is too expensive to store a separate SB-tree at every time instant. The features from the MVBT can be applied to reduce the space. While logically equivalent to a set of B+-trees, one at each time instant, the MVBT nicely embeds the set of B+-trees in such a way that the overall space is linear ([BGO+96]). Hence a Multiversion SB-tree satisfies the requirements we had set for the structure design.

4 The Multiversion SB-tree

The MVSBT is a new index that supports efficiently the insertion operation: "given key k, time t and value v, add v to the values associated with all the points in the rectangle [k, *maxkey*) x [t, *maxtime*)"; and the point query: "given key k and time t, find the value associated with this point in the key-time space".

4.1 Basic Idea

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

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

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

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

Since a record in the MVSBT has a rectangle (and not just a key range as it would be if we had kept an SB-tree for each time instant), the insertion algorithm needs to be modified accordingly. Assume the insertion of key k, time t and value v (represented as $\langle k, t\rangle : v$) goes into page p. All the fully-covered records in p should be split vertically at t (and by adding v to the value of the newly copied record). If there is a partly-covered record, the insertion algorithm should recursively insert into the partly-covered child page; at the leaf level, the partly-covered record is split into three (vertically at t and then horizontally at k, adding v to the top-right copy).

If an insertion causes a page to have more than b records, an $overflow$ occurs. All the alive records in the page is copied to a new page, and the *start* times of all the copied records are changed to the current insertion time. We call such an copy operation a *time split*. After a time split, the newly generated page may be almost full. In such a case, a few subsequent insertions in the page trigger a time split again, resulting a space cost of $\Theta(1)$ block per insertion. To avoid this phenomenon, we require that after a time split, the new block should have at most $f \cdot b$ records, where constant $f \in (0, 1)$ is called the *strong factor*. We call this requirement the *strong condition*. If a newly generated page due to a time split *strong overflows* (having more than $f \cdot b$ records), it is *key split*, that is, it is split into two (or more, if f is small) by key and the records are distributed evenly among these pages.

4.2 Optimizations

In this section we discuss three optimization techniques which apply to the MVSBT.

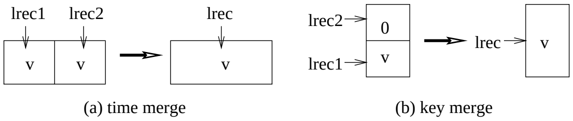
4.2.1 Aggregation in a Page

It is expensive to split all the fully-covered records in a page (each insertion introduces $\Theta(b)$ records). We propose an optimization technique which ensures that if there is no overflow, at most one ("representative") record is split in a page. The idea is that we only split the record with the smallest key range (the partly-covered record for a data page, or the first fully-covered record for an index page). This split physically adds a value v to only one record. We refer to this operation as ***logical splitting***. In order to deliver the correct response to a query, we have to modify the point query algorithm in the following way. A point query of $\langle k, t\rangle$ still aggregates the values of all the records containing the point along a path from root to leaf; but the value for each such record rec in page p is computed as the sum of all the records in p whose intervals contain t and whose ranges contain k or are lower than k.

This optimization also affects the key-split procedure. Before the key split of page p, the actual value of an alive record is computed as the sum of the values of all the alive records 'below' it (i.e., records having a smaller key range). If we key-split p into two pages, the sum of values of all the records in the page with the lower range should be added to the lowest record in the page with the higher range.

4.2.2 Record Merging

Record merging, if applicable allows to compact more records in a page and thus leads to less overall space. Two leaf records $lrec_1$, $lrec_2$ in the same page can be merged either horizontally (*time merge*) or vertically (*key merge*). A time merge can take place if (a) $lrec_1.range = lrec_2.range$; (b) $lrec_1.end = lrec_2.start$; and (c) $lrec_1.value = lrec_2.value$ (figure 2a). A key merge can take place if (a) $lrec_1.interval = lrec_2.interval$; (b) $lrec_1.high = lrec_2.low$; and (c) $lrec_2.value = 0$ (figure 2b).



(a) time merge

(b) key merge

Figure 2: Time merge and key merge of two records

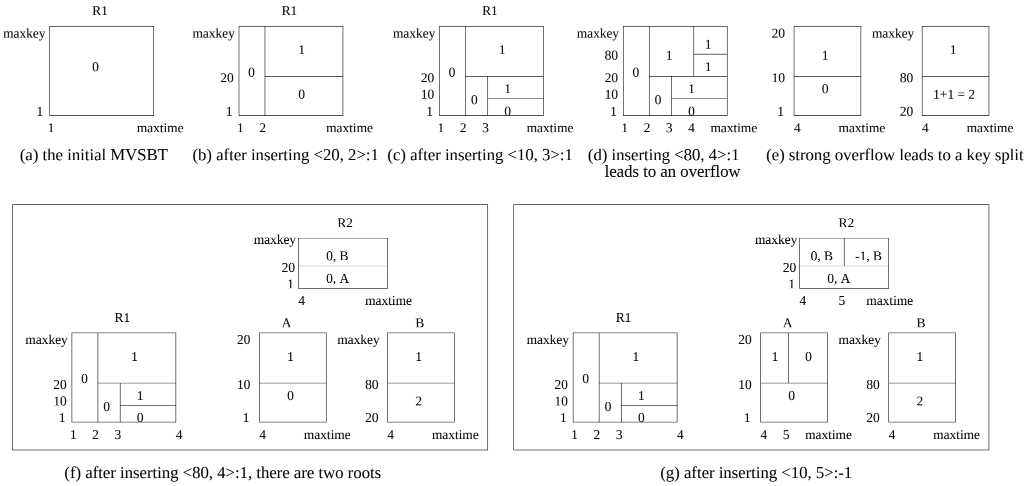
The index records can be merged similarly. The difference of merging index records from merging leaf records is that two index records can be merged only if they point to the same child page.

4.2.3 Page Disposal

Since we allow many insertions at the same time instant we should update the index about the "net" effect of these insertions. However, our algorithms process one update at a time. Hence we introduce the page-disposal optimization, which spares the index from "indermediate" results. If a page which is created at time t takes some subsequent insertions also at t and overflows, after the page is time split and key split, the page itself as well as the index record pointing to it can be physically removed from the index. This optimization saves space, too.

4.3 An Example

In this section, we assume $b = 6$ and $f = 0.5$. Initially, the MVSBT has one root page, R_1, which is also a leaf. There is one record in it having $value = 0$ (figure 3a). After we insert $\langle 20, 2\rangle : 1$, the record is split (figure 3b). To insert $\langle 10, 3\rangle : 1$, only the partly-covered record is split (figure 3c).
The insertion of $\langle 80, 4\rangle : 1$ causes an overflow (figure 3d). A time split copies all the alive records into a new page. If the new page satisfied the strong condition, it would be registered as the new root and the insertion would be complete. However, it strong overflows. So a key split takes place which distributes the records evenly into two pages (figure 3e). Note how the value of the first record in the page with higher range is modified. The tree after the insertion is shown in figure 3f. We now consider the insertion of $\langle 10, 5\rangle : -1$. In the alive root R_2, the first fully-covered record is split, and the insertion recursively goes to the partly-covered child page A. Since there is no partly-covered record in A, the first fully-covered record is split. The result is shown in figure 3g. Yet another insertion of $\langle 5, 5\rangle : 1$ would lead to a time merge in R_2 and a time merge in R_1. The figure is omitted due to space limitations.



(a) the initial MVSBT (b) after inserting <20, 2>:1 (c) after inserting <10, 3>:1 (d) inserting <80, 4>:1 leads to an overflow (e) strong overflow leads to a key split

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

(g) after inserting <10, 5>:-1

Figure 3: An example of insertions in an MVSBT

4.4 Complexity Analysis

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

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

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

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

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

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

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

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

Suppose there are n insertions in a MVSBT. Theorems 2 states the worst-case insertion cost, point query cost and the space complexity, respectively.

Theorem 2 *For a MVSBT, 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 $O(\frac{n}{b} \cdot \log_b K)$.*

A corollary of theorems 1 and 2 summarizes the performance of maintaining and computing the range-temporal aggregates as follows.

Corollary 1 *Using two MVSBTs, a SUM, AVG, COUNT RTA query is answered in $O(\log_b n)$ I/Os. The insertion/deletion 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 RTA query time is due to the time needed identifying the root of the appropriate SB-tree in the MVSBT graph. In practice, this search can be even faster if all different SB-tree roots created in the evolution are kept in a main-memory array, in which case the query time is reduced to traversing the appropriate SB-tree, i.e., $O(\log_b K)$.

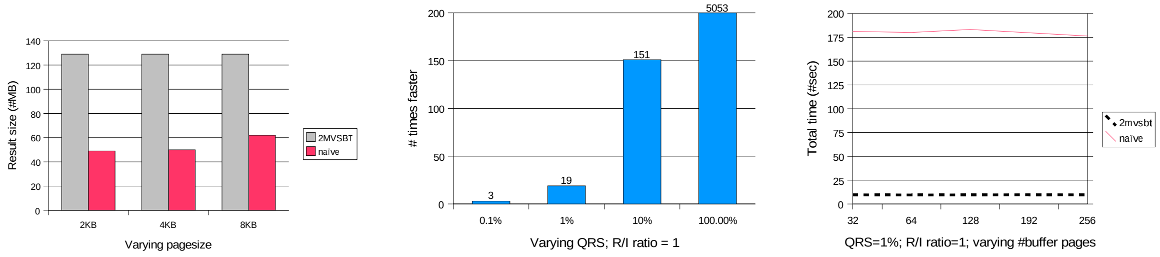
5 Performance Results

We present results comparing the performance of our approach with a naive approach where the temporal records are kept in a traditional temporal index, the MVBT ([BGO+96]).

The algorithms are implemented in C++ using GNU compilers. The programs run on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.8. The main memory size is 512 MB. To compare the performance of the various algorithms we use the estimated running time. This estimate is commonly obtained by multiplying the number of I/O's by the average disk page read access time, and then adding the measured CPU time. Following the practice in [APR+00], we measure the CPU cost by adding the amounts of time spent in *user* and *system* mode as returned by the *getrusage* system call. We assume all disk I/Os are random. A random disk access takes 10ms on average. We use a 4KB pagesize. For both MVSBT and MVBT we used LRU buffering and the default buffer size is 64 pages. The MVSBT uses a strong factor $f = 0.9$.

All the datasets we use were initially created using the TimeIT software ([KS98]) and then transformed to add record keys. We studied the effect of both uniformly distributed and normally distributed keys. Each dataset has 1 million records. The $key, start, end, value$ attributes of each record are all 4 bytes long. The key space is $[1,\ 10^6)$ and the time space is $[1,\ 10^8)$. A dataset contains 10,000 unique keys where on average there are 100 different records with the same key. We tested datasets with mainly long-lived intervals and with mainly short-lived intervals.

Figure 4a shows the space requirements for the MVBT and the two-MVSBT approach, for a dataset with uniformly distributed keys and with mainly long-lived intervals. The two-MVSBT approach used about 2.5 times more space than the single MVBT. This is to be expected, since the worst case space of each MVSBT has a $O(\log_b K)$ overhead. We observed a similar behavior for the update time per insertion/deletion as well.



(a) (b) (c)

Figure 4: Space and query performance comparisons

For the query performance we measured the execution time of 100 randomly generated query rectangles with fixed rectangle shape and size. The shape of a query rectangle is described by the *R/I ratio*, where R is the length of the query key range divided by the length of the key space and I is the length of the query time interval divided by the length of the time space. The *query rectangle size (QRS)* is described by the percentage of the area of the query rectangle in the whole key-time space.

Figure 4b shows how much faster the two-MVSBT approach is over the MVBT for the RTA query. Clearly, the larger the QRS is, the more advantageous the two-MVSBT approach is over the MVBT. When the query rectangle is the whole key-time space, the two-MVSBT is more than 5000 times faster than the naive approach! This is to be expected, since the query performance of the two-MVSBT is independent to the QRS, while the naive approach in the worst case scans the whole dataset. Figure 4c compares the query performance of QRS=1% of the key-time space over various buffer sizes. Again, the two-MVSBT approach is clearly superior.

6 Conclusions

Temporal aggregates have become predominant operators in analyzing historical data. This paper examines temporal aggregation queries in the presence of key-range predicates (RTA queries). Such queries allow the warehouse manager to focus on tuples grouped by some key range over a given time interval. We proposed a new index structure, the Multiversion SB-Tree (MVSBT), for incrementally maintaining and efficiently computing RTAs. The aggregates we considered are SUM, COUNT and AVG. The MVSBT has very fast (logarithmic) query time and update time, at the expense of a small space overhead. Initial performance results show the benefits of our solution. There are various interesting problems for further research: (i) how to optimize the performance of the MVSBT with factor f, (ii) how to support MIN/MAX temporal aggregate queries with range predicates, and, (iii) how to extend this work for spatiotemporal (i.e. multidimensional) aggregates.

References

[APR+00] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold and J. Vitter, "A Unified Approach For Indexed and Non-Indexed Spatial Joins", *Proc. of EDBT*, pp. 413-429, 2000.

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

[BKS+90] N. Bechmann, H. Kriegel, R. Schneider and B. Seeger, "The R* tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of ACM SIGMOD*, pp. 322-332, 1990.

[GHR+99] J. Gendrano, B. Huang, J. Rodrigue, B. Moon and R. Snodgrass, "Parallel Algorithms for Computing Temporal Aggregates", *Proc. of ICDE*, pp. 418-427, 1999.

[J+98] C. Jensen, et al, "The Consensus Glossary of Temporal Database Concepts - February 1998 Version", in *Temporal Databases: Research and Practice*, (ed.) O. Etzion, S. Jajodia and S. Sripada, Springer, ISBN 3-540-64519-5, pp. 367-405, 1998.

[KS95] N. Kline and R. Snodgrass, "Computing Temporal Aggregates", *Proc. of ICDE*, pp. 222-231, 1995.

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

[KTF98] A. Kumar, V. Tsotras and C. Faloutsos, "Designing Access Methods for Bitemporal Databases", *IEEE TKDE* 10(1), pp. 1-20, 1998.

[LS89] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", *Proc. of ACM SIGMOD*, pp. 315-324, 1989.

[MLI00] B. Moon, I. Lopez and V. Immanuel, "Scalable Algorithms for Large Temporal Aggregation", *Proc. of ICDE*, pp. 145-154, 2000.

[PS85] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin/Heidelberg, Germany, 1985.

[SS88] A. Segev and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment", *Proc. of Int. Conf. on Statistical and Scientific Database Management (SSDBM)*, pp. 39-61, 1988.

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

[YK97] X. Ye and J. Keane "Processing temporal aggregates in parallel", *Proc. of Int. Conf. on Systems, Man, and Cybernetics*, pp. 1373-1378, 1997.

[YW01] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", to appear in *Proc. of ICDE*, 2001. (also available at http://www-db.Stanford.EDU/~junyang/research/pubs.html)

A Insertion and Query Algorithms for the MVSBT

This appendix formally describes the insertion and point query algorithms for the MVSBT. To be clear, in the insertion algorithm we omit the details of the optimizations given in section 4.2.

Algorithm *PointQuery*(Key k, Time t)

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

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

```
$v = 0$;
for every record $rec$ in $p$ do
    if $rec$ is alive at $t$ and $rec.low \le k$ then
        $v = v + rec.value$;
    endif
endfor
if $p$ is a leaf page then
    return $v$;
else
    Find the record $rec$ whose rectangle contains $< k, t >$;
    return $v$ + PagePointQuery( $rec.child, k, t$);
endif
```

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

```
// Find the path of nodes containing partly covered records
$level = 0$;
$lowestpage$ = ReadPage(the latest root); // $ReadPage$ reads a page from disk if it is not in memory.
while $lowestpage$ is an index page and $lowestpage$ contains a partly-covered record $irec$, do
    $path[level] = lowestpage$;
    $level + +$;
    $lowestpage$ = ReadPage($irec.child$);
endwhile
// Handle lowestpage
if $lowestpage$ is a leaf page then
    if $lowestpage$ has enough space then
        if there is a partly-covered record then
            Split it in $lowestpage$;
        else
            Split in $lowestpage$ the first fully-covered record;
        endif
    else
        Copy alive leaf records from $lowestpage$ to $buffer$;
        if there is a partly-covered record then
            Split it in $buffer$;
        else
            Add $v$ to the first fully-covered record in $buffer$;
        endif
        Create new leaf pages (from records in $buffer$) and store their references in $toparent$;
    endif
else
    // $lowestpage$ is an index page that does not have partly-covered record
    // similar to the leaf page case; omit.
endif
// Handle the pages which contain partly-covered records bottom-up
for $x = level - 1$ downto 0 do
    if $path[x]$ has enough space then
        if $toparent$ is not empty then
```

```
            Insert records from $toparent$ to $path[x]$;
        endif
        Split the first fully-covered record in $path[x]$, if any;
    else
        Copy alive records from $path[x]$ to $buffer$;
        Add $v$ to the first fully-covered record in buffer, if any;
        Copy $toparent$ to $buffer$ if it is not empty;
        Create new index pages (from records in $buffer$) and store their references in $toparent$;
    endif
endfor
// Decide whether to create a new root page
if $toparent$ is not empty then
    Create a new root page from records in $toparent$;
endif
```

B Lemma and Theorem Proofs

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

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

Proof of 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*)

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

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

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

Let $p_1, p_2, \cdots, p_x$ be the longest successor path to p, i.e. $\forall i \in [1, x-1], p_{i+1}$ is a successor of p_i and $p_x = p$. Since p is not a root page, somewhere in the path there must be a key split. Let p_i be the result of the last key split which occur in the path. Suppose when p_i was about to be generated, there were $x \cdot f \cdot b - y$ records, where $x \geq 2$ and $0 \leq y < f \cdot b$. Right after p_i was generated, the number of records in it is at least $\lfloor \frac{x \cdot f \cdot b - y}{x} \rfloor = \lfloor f \cdot b - \frac{y}{x} \rfloor \geq \lfloor f \cdot b - \frac{f \cdot b - 1}{2} \rfloor = \lceil \frac{f \cdot b}{2} \rceil$.
Since in a page, the number of additions is no smaller than the number of deletions, for any time $t1$ before $p_i.end$, there are at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at $t1$. For all $j \in [i+1, x]$, when p_j is created, it has at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at $t1$ since there were at least this many to be copied from p_{j-1} and there is no strong overflow. For any later time before $p_j.end$, the number of alive records does not decrease. □

Proof of Lemma 4 (*The upper bound of the height of any sub-tree in a MVSBT is* $\lceil \log_{\lceil \frac{f \cdot b}{2} \rceil} (K+1) \rceil$):
Given a tree in an MVSBT. Consider each time instant $t \in$ the lifespan of the tree root. Since there are at most K different keys ever inserted in the tree, there are at most $K+1$ different leaf records which are alive at t. Since each leaf page alive at t contains at least $\lceil \frac{f \cdot b}{2} \rceil$ records alive at t, there are at most $\frac{K+1}{\lceil \frac{f \cdot b}{2} \rceil}$ leaf pages alive at t. This also means that there are at most this many index records which are alive at t and which point to these pages. So at one level up, there are at most $\frac{K+1}{\lceil \frac{f \cdot b}{2} \rceil^2}$ index pages alive at t. This argument is true for all levels until the root, where there is only one page alive at t. So there are at most $\lceil \log_{\lceil \frac{f \cdot b}{2} \rceil} (K+1) \rceil + 1$ levels. □

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

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

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

For the first set: We know that after a page is created, it takes at least $\mathrm{O}(b)$ insertions for it to overflow (lemma 2). So when a page overflows, there were at least $\mathrm{O}(b)$ insertions that went through this page after it was created. On the other hand, the overflow introduces at most $\mathrm{O}(b)$ occupied slots in the first set. So we can amortize the $\mathrm{O}(b)$ occupied slots to the $\mathrm{O}(b)$ insertions. Thus each insertion creates $\mathrm{O}(1)$ amortized copied slot for each page it goes through. Since an insertion goes through at most $\mathrm{O}(\log_b K)$ pages, an insertion creates $\mathrm{O}(\log_b K)$ slots in the first set as well.

To sum up, each insertion creates $\mathrm{O}(\log_b K)$ occupied slots. So for n insertions the total number of occupied slots is $\mathrm{O}(n \cdot \log_b K)$. Now we consider the minimum occupance 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 occupance of $O(b)$, too. So the total number of pages occupied by the SB-trees in an MVSBT is $\mathrm{O}(\frac{n}{b} \cdot \log_b K)$.

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