# Incremental Computation and Maintenance of Temporal Aggregates[*]

Jun Yang and Jennifer Widom
Computer Science Department, Stanford University
{junyang, widom}@db.stanford.edu

**Abstract**

*We consider the problems of computing aggregation queries in temporal databases, and of maintaining materialized temporal aggregate views efficiently. The latter problem is particularly challenging since a single data update can cause aggregate results to change over the entire time line. We introduce a new index structure called the* SB-tree*, which incorporates features from both* segment-trees *and* B-trees*. SB-trees support fast lookup of aggregate results based on time, and can be maintained efficiently when the data changes. We also extend the basic SB-tree index to handle* cumulative *(also called* moving-window*) aggregates. For materialized aggregate views in a temporal database or warehouse, we propose building and maintaining SB-tree indices instead of the views themselves.*

## 1. Introduction

*Temporal aggregation operators* are included in most temporal query languages, including TQuel [14] and TSQL2 [13]. Due to the rapidly increasing use of *data warehouses* to collect historical information, and the predominance of aggregation operators in analyzing this information, temporal aggregation is an important practical issue that has seen only moderate investigation to date (see Section 2). The efficient implementation of temporal aggregation operations, and the efficient management of temporal aggregate *views* such as those found in a data warehouse, present a number of unique challenges not found in the case of non-temporal aggregation.

One challenge is *temporal grouping*, a process in which we must group aggregate results by time. Consider for example the table $Prescription$ in Table 1, which stores prescription information for recipients of a certain drug. In temporal databases, each tuple is timestamped by a *valid* interval, indicating the time interval during which the tuple is "alive." Each $Prescription$ tuple records the name of the patient, daily dosage, and the prescription period (as the valid interval of the tuple). Let us assume that the granularity of time is one day, and for simplicity of presentation we use integers instead of actual dates for time instants. The contents of $Prescription$ are also illustrated graphically in Figure 1. Table 2 shows the contents of $SumDosage$, a temporal aggregate that computes the sum of active dosages along the time line. For example, the value of $SumDosage$ during the interval $[15, 20)$ is $6$, because there are three active prescriptions (Amy, Ben, and Fay) during $[15, 20)$, with a total daily dosage of $2 + 3 + 1 = 6$. At time 20, the aggregate value changes to 7 because Cal's prescription becomes active. As another example, Table 3 shows the contents of $AvgDosage$, a temporal aggregate that computes the average daily dosage along the time line. Clearly, computing these aggregate results is significantly more intricate than aggregation without the additional time dimension.

$SumDosage$ and $AvgDosage$ are termed *instantaneous* temporal aggregates because the value of these aggregates at a particular time instant is computed from the set of tuples that are valid at that instant. A further challenge is the computation of *cumulative* temporal aggregates [14]. A cumulative temporal aggregate has an additional parameter $w$ called the *window offset*. The value of a cumulative aggregate at time instant $t$ is computed over all tuples whose valid intervals overlap with the interval $[t - w, t]$. Intuitively, the result of a cumulative aggregate is a sequence of values generated by moving a window of given length along the time line, and evaluating the aggregate function over all tuples that are valid in the current window. (An instantaneous aggregate can be considered as a cumulative aggregate with window offset 0.) Table 4 shows the contents of $AvgDosage_5$, a cumulative aggregate that computes, at each point along the time line, the average of all dosages that were active at any point within the past 5 days. As another example, Table 5 shows the contents of $MaxDosage_{20}$, a cumulative aggregate that computes, at each point along the time line, the maximum of all dosages that were active at any point within the past 20 days. Cumulative aggregates such as $AvgDosage_5$ and $MaxDosage_{20}$ are even more complicated and expensive to compute than the instantaneous aggregates illustrated by $SumDosage$ and $AvgDosage$ earlier.

Now let us consider the problem of managing temporal aggregate views, particularly in a data warehousing context [16]. First, the warehouse should be able to maintain temporal aggregates *incrementally* as sources are updated. The alternative approach of recomputing temporal aggregates becomes progressively more inefficient as historical data accumulates, and in some cases it may even be impossible to recompute temporal aggregates because the ware-

[*]This work was supported by the National Science Foundation under grant IIS-9811947.

**Table 1:** $Prescription$**.**

| $patient$ | $dosage$ | $valid$ |
|---|---|---|
| "Amy" | 2 | $[10, 40)$ |
| "Ben" | 3 | $[10, 30)$ |
| "Cal" | 1 | $[20, 40)$ |
| "Dan" | 2 | $[5, 15)$ |
| "Eve" | 4 | $[35, 45)$ |
| "Fay" | 1 | $[10, 50)$ |

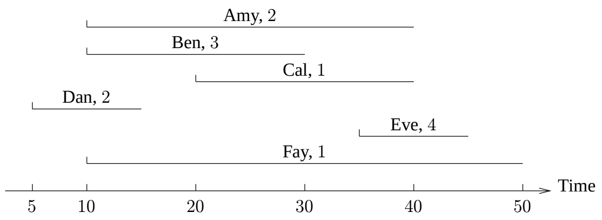

**Figure 1: Graphical representation of** $Prescription$**.**

house may not keep all of the historical data over which the aggregates are defined [18]. Another problem is that the traditional data warehousing approach of directly materializing and maintaining the view contents can be extremely inefficient for temporal aggregates. As an example, suppose we have materialized the contents of $SumDosage$ shown in Table 2. Now, suppose a tuple $\langle$"Guy", $5, [15, 45)\rangle$ is inserted into base table $Prescription$. To properly update $SumDosage$, we need to increment the value of $sum\_dosage$ by 5 for every tuple in $SumDosage$ whose valid interval is covered by $[15, 45)$; these are the third through the seventh tuples in Table 2. In other words, as the result of this insertion, more than half the tuples in $SumDosage$ must be updated. In general, when tuples with long valid intervals are inserted into or deleted from a base table, it is very expensive to update the contents of a temporal aggregate view over that table.

To recap, we have identified several problems related to temporal aggregation: (1) efficient computation of *instantaneous* temporal aggregates; (2) efficient computation of *cumulative* temporal aggregates; (3) maintaining temporal aggregate views *incrementally* to avoid expensive recomputation; and (4) the issue that even incremental maintenance can update large fragments of a temporal aggregate view. To address all of these problems, we introduce a new kind of index structure called the *SB-tree*. SB-trees are balanced, disk-based index structures that support fast lookups of temporal aggregate values by time. SB-trees also support efficient incremental updates, even when tuples with long valid intervals are inserted or deleted. Thus, rather than materializing and maintaining temporal aggregate views directly, we propose that SB-tree indices should be built and maintained instead. We also briefly outline our approach to handling cumulative temporal aggregation based on variations of SB-trees. Because of space constraints, details are presented in the full version of this paper [17].

## 2. Related Work

A first proposal for computing temporal aggregates was given in [15] and was based on an extension to the non-temporal aggregate computation algorithm from [2]. The approach consists of two steps, each requiring one scan of the base table. The first step determines the appropriate intervals for the aggregate result tuples, i.e., the partitioning of the time line into intervals, each with a constant aggregate value. The second step considers each tuple $t$ in the base table in turn, updating the aggregate values for all result tuples covered by $t$'s valid interval. Suppose that the size of the base table is $n$ and the number of result tuples is $m$. This approach has a worst-case running time of $O(mn)$, because a base tuple with a long valid interval can potentially contribute to $O(m)$ result tuples in the second step. Since the first step must be completed before the second one starts, this approach does not support incremental computation and maintenance of the aggregate results.

Moon et al. [10] proposed a *balanced-tree algorithm* based on red-black trees for computing temporal `SUM`, `COUNT`, and `AVG` aggregates. In the full paper [17], we generalize the balanced-tree algorithm so that it is not tied to any particular data structure, and call our generalized version the *endpoint sort algorithm*. The endpoint sort algorithm has the advantage that it can be implemented easily in a database system since sorting can be done by the database system without custom data structures. Both the balanced-tree and the endpoint sort algorithms have a worst-case running time of $O(n\log(m))$. For computing temporal `MIN` and `MAX` aggregates, Moon et al. proposed a *merge-sort algorithm* based on the divide-and-conquer strategy with a running time of $O(n\log(m))$. Unfortunately, none of these $O(n\log(m))$ algorithms supports incremental computation or maintenance of the aggregate results.

Moon et al. also presented a *bucket algorithm* for temporal aggregation and parallelized it on a shared-nothing architecture. The time line is partitioned into disjoint intervals, and tuples of the base table are partitioned accordingly based on their valid intervals; those with long valid intervals go into a *meta array*. Temporal aggregation can then be performed independently for each interval, using any algorithm. Results for all intervals are combined together and with the meta array. This algorithm is complementary to our approach and could be used to parallelize our algorithms.

Kline and Snodgrass [7] developed a data structure called the *aggregation tree* based on the binary *segment-tree* [11]. Aggregation trees support incremental computation of temporal aggregates. In particular, their segment-tree features allow efficient processing of tuples with long valid intervals. This point will be discussed in detail in Section 3, because our SB-trees also incorporate these segment-tree features. One drawback of the aggregation tree is that it is designed to be a main-memory data structure, which

**Table 2:** $SumDosage$.

| $sum\_dosage$ | $valid$ |
|---|---|
| 0 | $(-\infty, 5)$ |
| 2 | $[5, 10)$ |
| 8 | $[10, 15)$ |
| 6 | $[15, 20)$ |
| 7 | $[20, 30)$ |
| 4 | $[30, 35)$ |
| 8 | $[35, 40)$ |
| 5 | $[40, 45)$ |
| 1 | $[45, 50)$ |
| 0 | $[50, \infty)$ |

**Table 3:** $AvgDosage$.

| $avg\_dosage$ | $valid$ |
|---|---|
| `NULL` | $(-\infty, 5)$ |
| 2.00 | $[5, 30)$ |
| 1.75 | $[30, 35)$ |
| 2.00 | $[35, 40)$ |
| 2.50 | $[40, 45)$ |
| 1.00 | $[45, 50)$ |
| `NULL` | $[50, \infty)$ |

**Table 4:** $AvgDosage_5$.

| $avg\_dosage$ | $valid$ |
|---|---|
| `NULL` | $(-\infty, 5)$ |
| 2.00 | $[5, 20)$ |
| 1.75 | $[20, 35)$ |
| 2.00 | $[35, 45)$ |
| 2.50 | $[40, 50)$ |
| 1.00 | $[50, 55)$ |
| `NULL` | $[55, \infty)$ |

**Table 5:** $MaxDosage_{20}$.

| $max\_dosage$ | $valid$ |
|---|---|
| `NULL` | $(-\infty, 5)$ |
| 2 | $[5, 10)$ |
| 3 | $[10, 35)$ |
| 4 | $[35, 65)$ |
| 1 | $[65, 70)$ |
| `NULL` | $[70, \infty)$ |

limits its effectiveness as a database index and as a persistent data structure for maintaining temporal aggregates in a data warehousing environment. Another problem with the aggregation tree is that it is unbalanced. In the worst case, it takes $O(n^2)$ to compute a temporal aggregate from a base table with $n$ tuples, $O(n)$ to process an insertion into the base table, and $O(n)$ to perform a lookup of the aggregate value by time. To circumvent the problem, Kline and Snodgrass proposed a variant of the aggregation tree called the *$k$-ordered aggregation tree*, which takes advantage of the *$k$-orderedness* of the base table to enable garbage collection of tree nodes. However, garbage collection makes it impossible to use the aggregate tree as an index. Moreover, $k$-orderedness of a base table is difficult to measure in practice. In the worst case, the running time of the $k$-ordered-aggregation-tree algorithm is still $O(n^2)$, which could well be the case in a data warehousing environment where tuples are usually inserted in the order of their valid intervals. Parallel versions of the aggregation-tree algorithm are developed in [19, 4], but they all inherit the same limitations of the sequential version discussed above.

A lot of work has been done on indexing temporal data [12, 9]. Some temporal index structures use segment-trees. For example, Kolovson and Stonebraker [8] proposed the *SR-tree*, which combines the properties of the segment-tree and the *R-tree* [5]. However, segment-trees have never been used to index and maintain temporal aggregates.

None of the related work discussed above considers cumulative temporal aggregates. On the other hand, the dual SB-tree trick we use to handle cumulative temporal `SUM`, `COUNT`, `AVG` aggregates (Section 4) is quite reminiscent of the *prefix-sum* approach taken by Ho et al. [6] for computing range queries over data cubes. Maintaining precomputed prefix sums is expensive because each update to a cell in the data cube has a range effect on the prefix sums; in this sense, the two-dimensional case of the problem resembles temporal aggregate maintenance. To reduce the cost of updating prefix sums, Geffner et al. [3] proposed the *dynamic data cube*. The two-dimensional case of the dynamic data cube, called the *cumulative B-tree*, has similar performance characteristics as the SB-tree. However, the cumulative B-tree has a static structure determined by the size of the data cube, and in essence, it only handles updates with intervals of the form $(-\infty, t)$. In contrast, the SB-tree has a dynamic structure, and it handles updates with arbitrary intervals.

## 3. Instantaneous Temporal Aggregates

Let us begin by considering instantaneous temporal aggregates. We introduce our new index structure called the *SB-tree*. A separate SB-tree index is used for each aggregate we wish to compute and/or maintain. The SB-tree supports fast lookup of aggregate values by time, fast reconstruction of the aggregate over the entire time line, and efficient incremental update of the index structure.

The SB-tree incorporates features from both the *segment-tree* [11] and the *B-tree* [1]. The segment-tree features ensure that the index structure can be updated efficiently when base tuples with long valid intervals are inserted or deleted. The B-tree features ensure that the index structure is balanced and disk-efficient. Combining these features and adapting them to handle temporal aggregates requires us to develop new algorithms to search, update, balance, and compact an SB-tree. These algorithms will be discussed in detail in this section.

Intuitively, an SB-tree contains a hierarchy of intervals associated with partially computed aggregate results. There are three types of nodes in an SB-tree: the root node, the interior nodes, and the leaf nodes. All nodes have the same size. Each SB-tree has a *maximum branching factor* $b$ and a *maximum leaf capacity* $l$ which determine the layout of the SB-tree. Typically, $b$ and $l$ are chosen such that each SB-tree node fits exactly on one disk page. Here is a detailed description of the SB-tree index structure:

- An interior node can hold up to $b$ contiguous time intervals. At least $\lceil \frac{b}{2} \rceil$ of them are actually used, i.e., the node must be at least half full. Suppose that in an interior node $N$ (Figure 2) we want to represent $j$ time intervals $N.I_1, N.I_2, \ldots, N.I_j$. Then $j-1$ distinct time instants are stored in $N$ in ascending order. The $i$-th time instant, denoted $N.t_i$, terminates the $i$-th time interval $N.I_i$ and starts the $(i+1)$-st time interval $N.I_{i+1}$. Also, each interval in $N$ (say $N.I_i$) is associated with a *partial aggregate value* (denoted $N.v_i$) and a pointer to a child node (denoted $N.c_i$). For COUNT, SUM, MIN, and MAX aggregates, $N.v_i$ is a single numeric value. For AVG, $N.v_i$ is actually a pair of SUM and COUNT values, which, unlike a single AVG value, can be updated incrementally.

- A leaf node is similar to an interior node in structure, except a time interval in a leaf node is not associated with a pointer to a child node (Figure 3). A leaf node can accommodate up to $l$ contiguous time intervals, where at least $\lceil \frac{l}{2} \rceil$ time intervals are actually used.
- Typically, the root node is identical to an interior node in structure except that the root node is only required to have at least two time intervals (and hence two child nodes). In the special case where the root node is the only node in an SB-tree, the root node is identical to a leaf node in structure except that the root node is only required to have at least one time interval.
- For any non-leaf node $N$, consider the $i$-th time instant $N.t_i$. All time instants that appear in the subtree rooted at $N.c_i$ must be strictly less than $N.t_i$. All time instants that appear in the subtree rooted at $N.c_{i+1}$ must be strictly greater than $N.t_i$.

As a simple example, Figure 4 shows an SB-tree index for the aggregate *SumDosage* from Table 2 with $b = l = 4$. Details will be discussed below, and we will see more complicated examples later. Of course in practice, $b$ and $l$ are on the order of hundreds given any realistic disk page size, and $l$ may be up to 1.5 times as large as $b$ because there are no pointers to child nodes in leaves.

Next we provide a recursive interpretation for the time intervals represented in SB-tree nodes that handles the non-obvious end cases. Suppose node $N$ contains a total of $j$ time intervals. Consider the $i$-th time interval $N.I_i$. The start time of $N.I_i$, denoted $start(N.I_i)$, is specified below:

- If $i > 1$, then $start(N.I_i) = N.t_{i-1}$.
- If $i = 1$ and $N$ is the root, then $start(N.I_i) = -\infty$.
- If $i = 1$ and $N$ has a parent node $N'$ such that $N'.c_k = N$, then $start(N.I_i) = start(N'.I_k)$.

The end time, $end(N.I_i)$, is specified as follows:

- If $i < j$, then $end(N.I_i) = N.t_i$.
- If $i = j$ and $N$ is the root, then $end(N.I_i) = \infty$.
- If $i = j$ and $N$ has a parent node $N'$ such that $N'.c_k = N$, then $end(N.I_i) = end(N'.I_k)$.

Finally, $N.I_i$ is given by $\left[start(N.I_i), end(N.I_i)\right)$. For example, in Figure 4, the first interval of node $N_0$ is $(-\infty, 15)$, the second interval of $N_1$ is $[5, 10)$, the last interval of $N_3$ is $[40, 45)$, and the last interval of $N_4$ is $[50, \infty)$.

We now identify two useful properties of SB-trees. First, for any non-leaf node $N$, the $i$-th time interval $N.I_i$ is always the union of all time intervals in $N.c_i$. Second, the union of all time intervals found at the same level of an SB-tree is always $(-\infty, \infty)$, i.e., the entire time line.

### 3.1. Lookup

Suppose we have an SB-tree index and wish to find the value of the temporal aggregate at a given time instant. We search the SB-tree recursively, starting from the root, ending at a leaf, and accumulating the partial aggregate values along the way. In the following, we formally define the SB-tree lookup function $lookup(N, t)$, which searches the subtree rooted at node $N$ and returns an aggregate value for time instant $t$.

- In $N$, search for the time interval containing $t$. Suppose that this time interval is $N.I_i$.
- If $N$ is a leaf, then $lookup(N, t) = N.v_i$.
- If $N$ is not a leaf, then $lookup(N, t) = accum(N.v_i, lookup(N.c_i, t))$.

In the above, *accum* is a function that combines two aggregate values according to the type of the aggregate. The definition of *accum* is shown below. Recall that we treat an AVG aggregate value as a pair of SUM and COUNT values.

- For SUM and COUNT, $accum(x, y) = x + y$.
- For AVG, $accum(\langle x_{sum}, x_{count}\rangle, \langle y_{sum}, y_{count}\rangle) = \langle x_{sum} + y_{sum}, x_{count} + y_{count}\rangle$.
- For MIN, $accum(x, y) = \min(x, y)$.
- For MAX, $accum(x, y) = \max(x, y)$.

As an example, let us look up the value of the temporal aggregate *SumDosage* at time instant 19 using the SB-tree in Figure 4. We start with $lookup(N_0, 19)$ at the root node $N_0$. The second interval of $N_0$, $[15, 30)$, contains the time instant 19, points to node $N_2$, and has value 1. Hence, $lookup(N_0, 19) = 1 + lookup(N_2, 19)$, and we continue with $N_2$. The first interval of $N_2$, $[15, 20)$, contains 19 and has value 5. Since $N_2$ is a leaf, $lookup(N_2, 19) = 5$, so $lookup(N_0, 19) = 1 + 5 = 6$.

The SB-tree lookup function differs from B-tree lookup in that the result is not stored in one place; instead, the result must be calculated from the values stored in all nodes along the path from the root to the leaf. The additional calculation required does not increase the overall complexity of the lookup function: Both SB-tree and B-tree lookups have a running time of $O(h)$, where $h$ is the height of the tree.

### 3.2. Range Queries and Aggregate Reconstruction

An SB-tree index also can be used to answer range queries. In a range query, we are interested in the value of the temporal aggregate over a given time interval $I$. Since the aggregate value may change over time, the result of a range query is a table of tuples, where each tuple consists of an aggregate value and a subinterval of $I$. (This result is similar to the complete aggregate, e.g., Table 2, except $I$

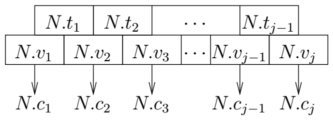

**Figure 2: An interior node.**

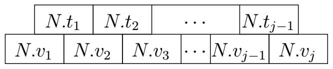

**Figure 3: A leaf node.**

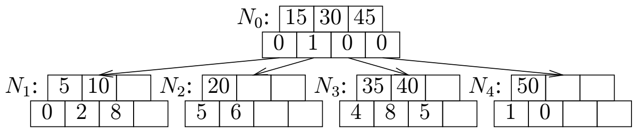

**Figure 4: SB-tree for $SumDosage$.**

need not be the entire time line.) To answer a range query, we perform a depth-first traversal (DFT) of the SB-tree to reach all leaf nodes containing time intervals that intersect with $I$. In the following, we formally define the procedure $range(N, I, v)$, which outputs the aggregate values together with their valid intervals during the time interval $I$ for the subtree rooted at node $N$. The third parameter $v$ is used to pass partially calculated aggregate values to recursive calls.

- If $N$ is a leaf, then for each $i$ such that $N.I_i \cap I \neq \varnothing$, output $\langle accum(N.v_i, v), N.I_i \cap I\rangle$.
- If $N$ is not a leaf, then for each $i$ such that $N.I_i \cap I \neq \varnothing$, call $range(N.c_i, I, accum(N.v_i, v))$.

In order to answer a range query over time interval $I$ using an SB-tree rooted at node $N_0$, we start with the call $range(N_0, I, v_0)$, where $v_0$ is an initial value defined according to the aggregate type:[1] For SUM and COUNT, $v_0 = 0$; for AVG, $v_0 = \langle 0, 0\rangle$; for MIN and MAX, $v_0 =$ NULL. The special value NULL has the the property that $accum(\texttt{NULL}, x) = accum(x, \texttt{NULL}) = x$ for any $x$.

For example, when executed on the SB-tree in Figure 4, $range(N_0, [14, 28), 0)$ returns the value of the temporal aggregate $SumDosage$ during $[14, 28)$. The nodes traversed by $range$ are $N_0$, $N_1$, and $N_2$. The output contains $\langle 8, [14, 15)\rangle$, $\langle 6, [15, 20)\rangle$, and $\langle 7, [20, 28)\rangle$, which correctly corresponds to Table 2.

To reconstruct the entire temporal aggregate from an SB-tree index, we simply run a range query using $I = (-\infty, \infty)$, which amounts to a DFT of the entire SB-tree. As an example, for the SB-tree in Figure 4, $range(N_0, (-\infty, \infty), 0)$ returns the contents of the temporal aggregate $SumDosage$ as shown in Table 2.

Range queries on SB-trees are processed differently from those on B-trees. Recall that in a B-tree (actually a B$^+$-tree to be specific), leaves are linked together in a sequence by pointers. To process a range query, we first search for the leaf containing the lower bound of the given range, and then follow pointers to find subsequent leaves within the range. The result values are all stored in leaves. In an SB-tree, however, result values cannot be obtained directly from the leaves; they must be calculated along the paths starting from the root. Therefore, we must use a DFT, which is why there is no need to link the leaves of an SB-tree together by pointers. Note that the DFT poses very little overhead in range query processing, especially when $b$ and $l$ are large. The running time of $range$ is proportional to the number of nodes traversed in the DFT, which is bounded by $O(h + r)$, where $h$ is the height of the SB-tree and $r$ is the number of leaves that intersect with the given interval. In other words, SB-tree range queries have the same asymptotic running time as B-tree range queries. As a corollary, the time required to reconstruct the entire temporal aggregate from an SB-tree is linear in the size of the aggregate.

[1] It is also acceptable to define $v_0 =$ NULL for SUM. In that case, if there is no base tuple valid at time instant $t$, the value of SUM at $t$ will be NULL instead of 0. This change will not affect any of our algorithms.

## 3.3. Insertion

Whenever a tuple is inserted into a base table, we need to update the SB-tree index for any temporal aggregate defined over this base table. Recall that the SB-tree indexes the aggregate and not the base table. Hence, unlike an insertion into a B-tree, which typically results in an additional entry in the tree for the new tuple, an insertion usually results in updates to various parts of the SB-tree, which reflect the effect of the new base tuple on the aggregate.

Consider inserting a tuple $t$ into a base table. Suppose that the value of $t$'s aggregated attribute is $v_{base}$, and $t$ is valid during the time interval $I$. The effect of this insertion on an aggregate can be captured by a pair $\langle v, I\rangle$, where $v$ is defined according to the type of the aggregate: For SUM, MIN, and MAX, $v = v_{base}$; for COUNT, $v = 1$; for AVG, $v = \langle v_{base}, 1\rangle$. In the following, we formally define the procedure $insert(N, \langle v, I\rangle)$, which updates the subtree rooted at node $N$ in order to process an insertion whose effect on the aggregate is $\langle v, I\rangle$. For each $i$ such that $N.I_i \cap I \neq \varnothing$:

- ∘ If $N.v_i = accum(v, N.v_i)$, do nothing.
- ∘ Otherwise, if $N.I_i \subseteq I$, set $N.v_i$ to $accum(v, N.v_i)$.
- ∘ Otherwise, $N.I_i \not\subseteq I$.
  - ◇ If $N$ is not a leaf, call $insert(N.c_i, \langle v, I\rangle)$.
  - ◇ If $N$ is a leaf, update $N$ to reflect the effect of $\langle v, I\rangle$.

There are a number of subtleties in the above procedure. First, note that the recursion stops before $N.c_i$ if the insertion has no effect on $N.v_i$. This check is primarily for MIN and MAX aggregates. In the case of MIN, for example, $N.v_i$ is an upper bound for the aggregate value during the interval $N.I_i$, because a lookup of the aggregate value anywhere during $N.I_i$ will pass through $N$ and see $N.v_i$. Therefore, if $v$ is already greater than $N.v_i$, the insertion cannot have any effect on the subtree rooted at $N.c_i$. The case of MAX is symmetric. This check can eliminate many unnecessary recursive steps from the $insert$ procedure.

Second, note that if $N.I_i$ is contained in $I$, we simply update $N.v_i$ and then stop, without further recursing down

$N.c_i$. Both *lookup* and *range* still can see the effect of this insertion because they accumulate all partial aggregate values along the path of traversal. This feature of the SB-tree, borrowed from the segment-tree, ensures that tuples with long valid intervals can be inserted efficiently. For example, if we insert tuple $\langle$“Guy”$, 5, [15, 45)\rangle$ into the *Prescription* table in Table 1, only $N_0.v_1$ and $N_0.v_2$ in Figure 4 need to be incremented by 5. Without this segment-tree feature, every leaf interval in $N_1$ and $N_2$ would need to be updated. The saving may seem insignificant for this simple example, but for larger, more realistic examples, the saving will be quite substantial if we can avoid updating entire subtrees.

The last line of the *insert* procedure, updating a leaf, is best illustrated with an example. If we insert $\langle$“Hal”$, 1, [24, 30)\rangle$ into *Prescription*, node $N_2$ in Figure 4 will contain one more interval. The old interval $N_2.I_2 = [20, 30)$ with value $N_1.v_2 = 6$ will be divided into two intervals: $[20, 24)$ with value 6, and $[24, 30)$ with value 7. Had we inserted $\langle$“Hal”$, 1, [24, 28)\rangle$ instead, $N_2.I_2$ would be divided into three intervals: $[20, 24)$ with value 6, $[24, 28)$ with value 7, and $[28, 30)$ with value 6. In general, an insertion can result in up to two more intervals in a leaf, possibly causing the leaf to overflow. In Section 3.5, we will show how to split nodes in order to deal with overflows.

As a slightly more complicated example, suppose that we insert $\langle$“Ida”$, 1, [17, 47)\rangle$ into *Prescription*. We execute $insert(N_0, \langle 1, [17, 47)\rangle)$ on the SB-tree in Figure 4. At node $N_0$, we examine the three intervals $N_0.I_2$, $N_0.I_3$, and $N_0.I_4$, which overlap with $[14, 47)$. $N_0.I_2 = [15, 29)$ is not completely covered by $[17, 47)$, so we continue with $insert(N_2, \langle 1, [17, 47)\rangle)$. $N_0.I_3 = [30, 45)$ is completely covered by $[17, 47)$, so we simply increment $N_0.v_3$ by 1. $N_0.I_4 = [45, \infty)$ is not completely covered by $[17, 47)$, so we continue with $insert(N_4, \langle 1, [17, 47)\rangle)$. We omit the details of calling *insert* on $N_2$ and $N_4$. The resulting SB-tree is shown in Figure 5.

All nodes examined by $insert(N, \langle v, I\rangle)$ lie either on the path from the root to the node covering the beginning of $I$, or on the path from the root to the node covering the end of $I$. Any node outside the region bounded by these two paths need not be examined because it contains no intervals that overlap with $I$. Any node within the region bounded by the two paths need not be examined either, because all its intervals are completely covered by some interval in an ancestor node that lies on one of the two paths. Therefore, the running time of *insert* is $O(h)$, where $h$ is the height of the SB-tree. This analysis does not yet take node splitting into account; a thorough analysis will be provided in Section 3.6.2.

### 3.4. Deletion

It is well known that `MIN` and `MAX` aggregates in general are not incrementally maintainable when tuples are deleted from the base table, a problem that is not specific to temporal aggregates. Hence, in this section, we focus on how to handle deletions for `SUM`, `COUNT`, and `AVG` aggregates.

The technique is simply to treat a deletion as an insertion with a “negative” effect on the aggregate value. Consider deleting a tuple $t$ from a base table. Suppose that the value of $t$'s aggregated attribute is $v_{base}$, and $t$ is valid during the time interval $I$. The effect of this deletion on an aggregate can be captured by a pair $\langle v, I\rangle$, where $v$ is defined according to the type of the aggregate: For `SUM`, $v = -v_{base}$; for `COUNT`, $v = -1$; for `AVG`, $v = \langle -v_{base}, -1\rangle$. Then, the deletion is handled by calling $insert(N, \langle v, I\rangle)$, where $N$ is the root of the SB-tree to be updated. As we have seen in Section 3.3, the running time of this procedure is $O(h)$, where $h$ is the height of the SB-tree.

For example, consider deleting $\langle$“Ida”$, 1, [17, 47)\rangle$ which we just inserted into *Prescription* in Section 3.3. Following the procedure $insert(N_0, \langle -1, [17, 47)\rangle)$ on the SB-tree in Figure 5, we obtain the SB-tree in Figure 6. Notice that Figure 6 is not identical to Figure 4, the SB-tree before the insertion and the deletion. In particular, the first and the second intervals of $N_2$ in Figure 6 have the same aggregate value; so do the first and the second intervals of $N_4$. These adjacent intervals with equal aggregate values can and should be merged. In Section 3.6, we will show how to merge such intervals to compact the SB-tree.

### 3.5. Node Splitting

As we have seen in Section 3.3, a leaf may become one or two intervals too full as the result of an insertion. When overflow occurs, we split the leaf into two leaves, each of which is roughly half full. Then, we need to split the corresponding interval in the parent node into two intervals, and associate them with the two new leaves. As a consequence, the parent node could overflow, so we may need to continue the splitting process up the SB-tree. Finally, if the root overflows, we split it into two and create a new root to point to them. The detailed node splitting procedure *split* is specified in the full version of this paper [17].

For example, consider executing $insert(N_0, \langle 1, [7, 12)\rangle)$ on the SB-tree in Figure 4. The resulting SB-tree before any node splitting is shown in Figure 7. Node $N_1$ overflows, so we split $N_1$ into $N_{11}$ and $N_{12}$, and we also split the first interval of $N_0$ at time instant 10. The resulting SB-tree is shown in Figure 8. Now $N_0$ overflows. Hence, we further split $N_0$ into $N_{01}$ and $N_{02}$, and then create a new root $N_0'$ to point to $N_{01}$ and $N_{02}$. The final result is shown in Figure 9.

The *split* procedure is invoked for each overflowing leaf in the SB-tree after an insertion or a deletion. Each insertion or deletion can cause at most two leaves to overflow. Since all splitted nodes must lie on the same path from the root, the running time of *split* is $O(h)$, where $h$ is the height of the SB-tree. Because *insert* itself takes $O(h)$, the overall time to process an insertion or a deletion is still $O(h)$.

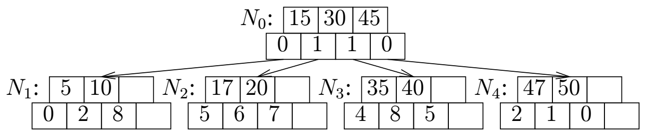

**Figure 5: SB-tree after *insert*.**

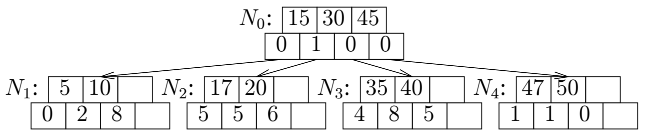

**Figure 6: SB-tree after *delete*.**

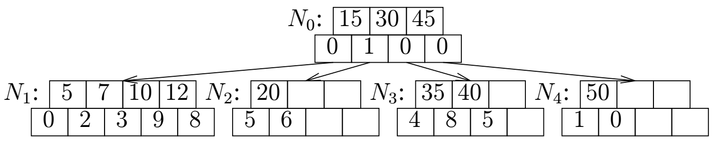

**Figure 7: SB-tree before $split(N_1)$.**

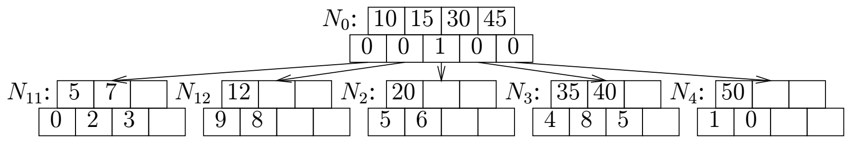

**Figure 8: SB-tree before $split(N_0)$.**

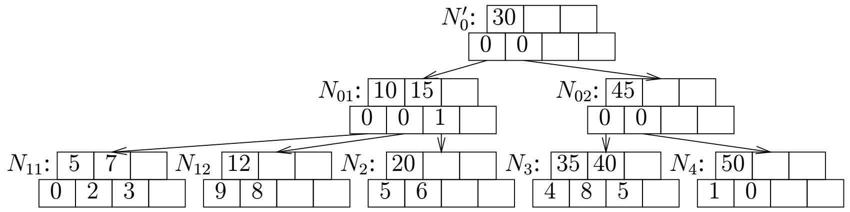

**Figure 9: SB-tree after *split* completes.**

## 3.6. Interval and Node Merging

At this point, it might appear that we have complete procedures to handle insertions and deletions, but in fact one subtlety remains. Both insertions and deletions are handled by *insert* and *split*, neither of which ever shrinks the SB-tree. A monotonically growing SB-tree is certainly unacceptable; we need a way of compacting it.

In Section 3.4, we saw that a deletion may result in two adjacent leaf intervals with equal aggregate values (Figure 6). In fact, an insertion could produce the same effect. For instance, in the example of Section 3.4, we could have inserted a tuple $\langle$"Jay", $-1, [17, 47)\rangle$ into *Prescription* instead of deleting the tuple $\langle$"Ida", $1, [17, 47)\rangle$ and obtained exactly the same SB-tree as in Figure 6. We can merge adjacent intervals with equal aggregate values, at which point a node may become less than half full. To deal with an underfull node, we can either move intervals from its sibling or merge it with its sibling.

The interval merging procedure *imerge* is used to merge two adjacent leaf intervals with equal aggregate values. The detailed *imerge* procedure is specified in the full paper [17]. The two adjacent intervals may belong to the same leaf, in which case they have equal aggregate values if and only if they have equal partial aggregate values in the leaf. The second case is more complicated: One interval is the last in a leaf, and the other interval is the first in the next leaf. In this case, we must start from their common ancestor and traverse down to these two intervals, accumulating the partial aggregate values along the two paths respectively. If the two results are equal, then the two intervals have equal aggregate values. There is no need to check any partial aggregate values above the common ancestor because they are shared by both leaf intervals.

The *imerge* procedure results in a leaf with one fewer interval. If this leaf has become less than half full, we call the node merging procedure *nmerge*. In general, if a non-root node $N$ is less than half full, $nmerge(N)$ attempts to move an interval from a sibling that contains more than the minimum number of intervals. If no sibling of $N$ has a "spare" interval, $nmerge(N)$ will merge $N$ with a sibling, and then merge their corresponding intervals in their parent node. As a result, the parent node could become underfull, so we may need to continue the process up the SB-tree. Finally, we may remove the root if it only has one interval left. Although this high-level description of *nmerge* is short, the details are quite involved because we must manipulate partial aggregate values stored in the interior nodes carefully in order to ensure that every transformation of the SB-tree preserves the value returned by *lookup* along every path. Again, because of space constraints, the procedure $nmerge(N)$ is specified in full in [17].

As a simple example, consider the SB-tree in Figure 6. We run *imerge* twice, first on the first and the second intervals of $N_2$, and then on the first and the second intervals of

$N_4$. The final result is identical to the SB-tree in Figure 4. In this example, *nmerge* is not needed.

As a more complicated example, let us continue with the example in Section 3.5. First, we delete the newly inserted tuple by running $insert(N_0, \langle -1, [7, 12) \rangle)$ on the SB-tree in Figure 9; the result is shown in Figure 10. We call *imerge* for the second and the third intervals of $N_{11}$, and for the first and the second intervals of $N_{12}$, since they are pairs of adjacent intervals with equal aggregate values. Figure 11 shows the state of the SB-tree right before we call $nmerge(N_{12})$ because node $N_{12}$ has become too small. Since both siblings of $N_{12}$ contain no spare intervals, $nmerge(N_{12})$ proceeds to merge $N_{12}$ with one of its siblings, say $N_2$, into a new node $N_2'$. At the same time, it merges the second and the third intervals of the parent node $N_{01}$. The final result is shown in Figure 12. Notice that the SB-tree in Figure 12 is not identical to the one we started with in Figure 4. Nevertheless, they encode exactly the same aggregate.

More comprehensive examples can be found in [17]. If we remove all tuples that have been inserted into the base table, the SB-tree will eventually become empty through interval and node merging. In general, an empty SB-tree only has a root node containing a single interval $(-\infty, \infty)$ with an initial aggregate value $v_0$ as defined in Section 3.2.

### 3.6.1. When to Compact

A compact SB-tree has the property that no two adjacent leaf intervals have the same aggregate value. In other words, if we perform *range* on a compacted SB-tree over $(-\infty, \infty)$, we will get a "normalized" result that cannot be represented by fewer tuples (without overlapping intervals). On the other hand, if an SB-tree is not compact, it will contain more leaf intervals than necessary, and *range* over $(-\infty, \infty)$ will output consecutive tuples with equal aggregate values. A compact SB-tree is desirable because its (potentially) lower height leads to more efficient operations.

Ideally, to ensure the compactness of an SB-tree after an insertion or deletion, we should perform *imerge* for each pair of adjacent leaf intervals with equal aggregate values. First, we must be able to detect such intervals. Recall that in order to calculate the aggregate value for a leaf interval, we must traverse all the way down to the leaf. Let $I$ denote the interval affected by the insertion or deletion. If we check every leaf interval that intersects with $I$, the overhead would completely negate the advantage of segment-tree features in handling base tuples with long valid intervals. To avoid this problem, we take one of two approaches depending on the type of the aggregate.

For `SUM`, `COUNT`, and `AVG`, $I$'s two endpoints will become interval endpoints in the SB-tree, and it suffices to check the two pairs of leaf intervals surrounding $I$'s two endpoints. Usually, each pair belongs to a single leaf, in which case we only need to compare the partial aggregate values in the leaf. In the worst case, two intervals in a pair may lie on two almost disjoint paths from the root, so the time it takes to perform the check is $O(h)$, where $h$ is the height of the SB-tree. There is no need to check intervals within $I$: If two adjacent intervals within $I$ had different aggregate values before the update, then they must have different aggregate values after the update, because all aggregate values within $I$ are incremented or decremented uniformly by the update. Since the common case carries very little overhead and the worst case does not increase the asymptotic complexity of the update operation, we can afford to keep the SB-tree compact at all times for `SUM`, `COUNT`, and `AVG`.

For `MIN` and `MAX`, it is possible for any two adjacent leaf intervals to have equal aggregate values after an update. For example, two adjacent leaf intervals with `MIN` values 2 and 3, respectively, will be updated to have the same `MIN` value of 1 when we insert a tuple with value 1 whose valid interval covers both of the leaf intervals. We still want to avoid the overhead of checking every leaf interval within $I$. Therefore, instead of calling *imerge* after every *insert* call on a `MIN` or `MAX` SB-tree, we periodically compact the SB-tree with a batch procedure *bmerge*. This procedure performs *range* on the SB-tree over $(-\infty, \infty)$, and combines output tuples with equal aggregate values and adjacent valid intervals. As soon as *bmerge* generates a tuple, it inserts the tuple into a second, initially empty SB-tree, which eventually replaces the original SB-tree as the index for the aggregate.

### 3.6.2. Complete Update Running Time

For `SUM`, `COUNT`, and `AVG`, the complete SB-tree update procedure includes calls to *insert*, *split*, and/or *imerge*/*nmerge* for up to two pairs of adjacent intervals. Both *insert* and *split* have a running time of $O(h)$, where $h$ is the height of the SB-tree. As we have shown in Section 3.6.1, checking for adjacent intervals with equal aggregate values requires $O(h)$. The running time of interval and node merging is dominated by the running time of *nmerge*, which is also $O(h)$ because the height of the tree limits the depth of the recursion in *nmerge*. In summary, the complete procedure takes $O(h)$. Furthermore, since the SB-tree is kept compact at all times, $O(h) = O(\log m)$, where $m$ is the number of tuples in the normalized aggregate result.

For `MIN` and `MAX`, each SB-tree update, including calls to *insert* and *split* but not *imerge*/*nmerge*, still has a running time of $O(h)$. Since the SB-tree is not kept compact at all times, in the worst case $O(h) = O(\log n)$, where $n$ is the total number of *insert* calls performed on the SB-tree (or, equivalently, the size of the base table; recall that we do not handle deletions for `MIN` and `MAX` aggregates). Note that $O(\log n)$ is not as efficient as $O(\log m)$ because the number of tuples in the aggregate result might be significantly less than the number of tuples in the base table. A possible optimization is to compact the SB-tree periodically using *bmerge*, whose running time is $O(n + m \log m)$.

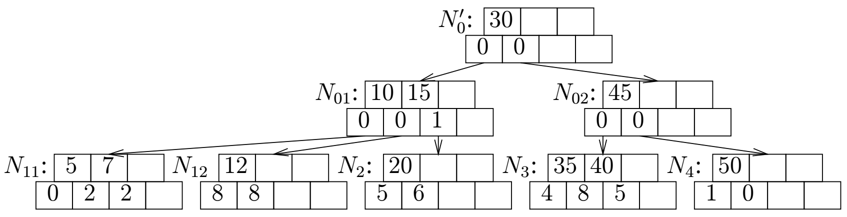

**Figure 10: SB-tree before $imerge$.**

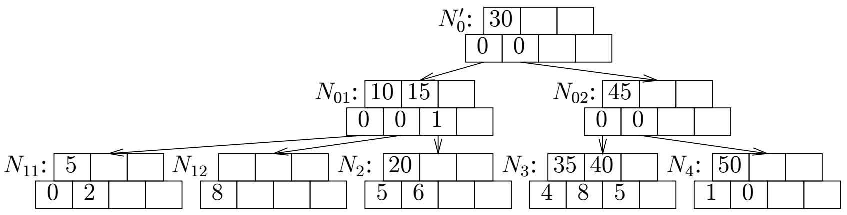

**Figure 11: SB-tree before $nmerge(N_{12})$.**

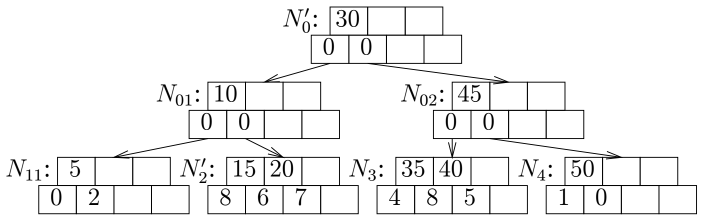

**Figure 12: SB-tree after $imerge$ completes.**

## 4. Cumulative Temporal Aggregates

In this section we briefly outline our approach to handling cumulative temporal aggregates. For details, please refer to the full version of this paper [17]. Recall from Section 1 that a cumulative temporal aggregate is computed with an additional parameter $w$, for window offset. If $w$ is fixed and known in advance, the solution is straightforward. We keep a separate SB-tree for the particular value of $w$. This SB-tree operates exactly like a regular SB-tree, except that an insertion with effect $\langle v, I\rangle$ on an instantaneous aggregate, where $I = [I_{start}, I_{end})$, is treated as an insertion with effect $\langle v, [I_{start}, I_{end} + w)\rangle$.

Supporting cumulative aggregates with arbitrary window offsets is more challenging. For SUM, COUNT, and AVG, we use a solution called *dual SB-trees*. The solution maintains a second SB-tree $T'$ (rooted at $N'$), in addition to the SB-tree $T$ (rooted at $N$) for the instantaneous aggregate. Recall that $lookup(N, t)$ returns an aggregate value computed over all base tuples that are valid at time instant $t$. We construct $T'$ in such a way that $lookup(N', t)$ returns an aggregate value computed over all tuples that are valid strictly before $t$. Then, the value of the cumulative aggregate with window offset $w$ at time $t$ can be computed as the difference between $lookup(N', t)$ and $lookup(N', t - w)$, adjusted by $lookup(N, t)$. The details of this and other operations on dual SB-trees can be found in [17]. In [17] we also consider an alternative called the *JSB-tree*, which provides interesting performance trade-offs with dual SB-trees.

Unlike SUM, COUNT, and AVG, it is possible to compute a cumulative MIN or MAX aggregate with arbitrary window offset from the SB-tree constructed for the corresponding instantaneous aggregate. To find the value of the cumulative aggregate with window offset $w$ at time instant $t$, we simply call $range(N, [t - w, t], v_0)$, where $v_0$ is the value defined in Section 3.2; the answer we are looking for is the MIN or MAX value of all the output tuples. When $w$ is large, however, this lookup operation may be too slow. To reduce the cost of lookup, we can store and maintain MIN or MAX values for subtrees inside non-leaf nodes. We call the resulting new index structure an *MSB-tree* (for MIN/MAX SB-tree), described in detail in [17].

Compared to the basic SB-tree, dual SB-trees and MSB-trees require only a small, constant factor more storage and running time for their operations, and they are able to handle cumulative aggregates with arbitrary window offsets not known in advance.

## 5. Conclusion

We have presented a new index structure for temporal aggregation called the SB-tree. SB-trees and their variants provide a number of improvements over previous approaches to implementing temporal aggregates. In Table 6, we compare our algorithms with the other temporal aggregation algorithms discussed in Section 2. For simplicity, Table 6 provides only rough upper bounds on the running time; please refer to corresponding sections or the full pa-

Table 6: **Comparison of temporal aggregation algorithms ($n$ is the size of the base table).**

| | aggregates handled | memory-based or disk-based | time to compute full aggregate | incrementally maintainable (update time) | usable as index (lookup time) | support for cumulative aggregates |
|---|---|---|---|---|---|---|
| basic [15] | all | disk | $O(n^2)$ | no | no | no |
| balanced tree [10] | `SUM`/`COUNT`/`AVG` | memory | $O(n \log n)$ | no | no | no |
| endpoint sort (see full version [17]) | `SUM`/`COUNT`/`AVG` | disk | $O(n \log n)$ | no | no | no |
| merge sort [10] | `MIN`/`MAX` | disk | $O(n \log n)$ | no | no | no |
| aggregation tree [7] | all | memory | $O(n^2)$ | $O(n)$ | $O(n)$ (no if $k$-ordered) | no |
| SB-tree (Sections 3 and 4) | all | disk | $O(n \log n)$ | $O(\log n)$ | $O(\log n)$ | fixed window offset |
| dual SB-trees (Section 4) | `SUM`/`COUNT`/`AVG` | disk | $O(n \log n)$ | $O(\log n)$ | $O(\log n)$ | arbitrary window offset |
| MSB-tree (Section 4) | `MIN`/`MAX` | disk | $O(n \log n)$ | $O(\log n)$ | $O(\log n)$ | arbitrary window offset |

per [17] for detailed analyses.

As future work, we plan to implement the SB-tree and its variants and measure their performance with real-world applications. We also plan to consider concurrency control algorithms for these index structures.

## References

[1] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[2] R. Epstein. Techniques for processing of aggregates in relational database systems. Technical Report UCB/ERL M7918, University of California, Berkeley, California, 1979.

[3] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. of the 2000 Intl. Conf. on Extending Database Technology*, pages 237–253, Konstanz, Germany, March 2000.

[4] J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel algorithms for computing temporal aggregates. In *Proc. of the 1999 Intl. Conf. on Data Engineering*, pages 418–427, Sydney, Australia, March 1999.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, Boston, Massachusetts, June 1984.

[6] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 73–88, Tucson, Arizona, June 1997.

[7] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pages 222–231, Taipei, Taiwan, March 1995.

[8] C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 138–147, Denver, Colorado, May 1991.

[9] H.-P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 407–418, Cairo, Egypt, September 2000.

[10] B. Moon, I. F. V. Lopez, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 145–154, San Diego, California, March 2000.

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

[12] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 21(2):158–221, June 1999.

[13] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[14] R. T. Snodgrass, S. Gomez, and L. E. McKenzie. Aggregates in the temporal query language TQuel. *IEEE Trans. on Knowledge and Data Engineering*, 5(5):826–842, October 1993.

[15] P. A. Tuma. Implementing historical aggregates in TempIS. Master's thesis, Wayne State University, Detroit, Michigan, 1992.

[16] J. Yang and J. Widom. Maintaining temporal views over non-temporal information sources for data warehousing. In *Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 389–403, Valencia, Spain, March 1998.

[17] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. Technical report, Stanford University, Stanford, California, August 2000. Available at `www.db.stanford.edu/~junyang/yw-tempaggr.ps`.

[18] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. In *Proc. of the 2000 Intl. Conf. on Extending Database Technology*, pages 395–412, Konstanz, Germany, March 2000.

[19] X. Ye and J. A. Keane. Processing temporal aggregates in parallel. In *Proc. of the 1997 IEEE Intl. Conf. on Systems, Man, and Cybernetics*, pages 1373–1378, Orlando, Florida, October 1997.