Incremental Computation and Maintenance of Temporal Aggregates*

Jun Yang and Jennifer Widom
Computer Science Department, Stanford University
{junyang,widom}@db.stanford.edu, http://www-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 extend the basic SB-tree index to handle *cumulative* (also called *moving-window*) aggregates, considering separately cases when the window size is or is not fixed in advance. For materialized aggregate views in a temporal database or warehouse, we propose building and maintaining SB-tree indices instead of the views themselves.

Category: research.

1 Introduction

Temporal aggregation operators are included in most temporal query languages, including TQuel [SGM93] and TSQL2 [Sno95]. 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 Figure 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 2. Figure 3 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, Figure 4 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.

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

$patient$	$dosage$	$valid$
"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: $Prescription$.

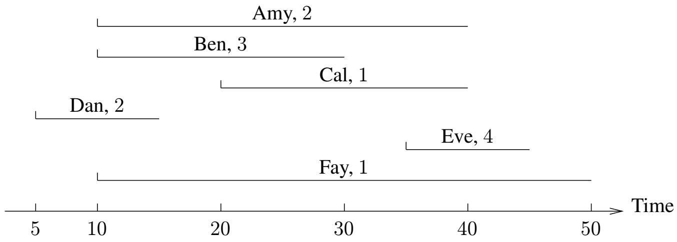

Figure 2: Graphical representation of $Prescription$.

$SumDosage$ 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 [SGM93]. 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 thought of as a cumulative aggregate with window offset 0.) Figure 5 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, Figure 6 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 [YW98]. 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 warehouse may not keep all of the historical data over which the aggregates are defined [YW00]. 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 Figure 3. 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 Figure 3. 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 introduced 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; (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.

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	$[50, \infty)$

Figure 3: $SumDosage$.

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	$[50, \infty)$

Figure 4: $AvgDosage$.

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	$[55, \infty)$

Figure 5: $AvgDosage_5$.

max_dosage	$valid$
NULL	$(-\infty, 5)$
2	$[5, 10)$
3	$[10, 35)$
4	$[35, 65)$
1	$[65, 70)$
NULL	$[70, \infty)$

Figure 6: $MaxDosage_{20}$.

Thus, rather than materializing and maintaining temporal aggregate views directly, we propose that SB-tree indices should be built and maintained instead. We also show how to adapt an SB-tree index to support cumulative temporal aggregation when there is a fixed window offset known in advance. A more difficult challenge is supporting cumulative aggregates with arbitrary window offsets not known in advance. We show how to use a pair of SB-trees to support cumulative `SUM`, `COUNT`, and `AVG` aggregates with arbitrary window offsets. We also introduce a second index structure called the *MSB-tree*, an extension to the SB-tree index that supports cumulative `MIN` and `MAX` aggregates with arbitrary window offsets.

2 Related Work

A first proposal for computing temporal aggregates was given in [Tum92] and was based on an extension to the non-temporal aggregate computation algorithm from [Eps79]. 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 two steps are separate and the first one must be completed before the second starts, this approach does not support incremental computation and maintenance of the aggregate results.

[MLI00] proposed a *balanced-tree algorithm* based on red-black trees for computing temporal `SUM`, `COUNT`, and `AVG` aggregates. In Section 5, 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, [MLI00] 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.

[MLI00] 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.

[KS95] developed a data structure called the *aggregation tree* based on the binary *segment-tree* [PS85]. 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 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, [KS95] 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. [YK97] and [GHR$^+$99] developed parallel versions of the aggregation-tree algorithm, but they all inherit the same limitations of the sequential version discussed above.

Finally, none of the related work discussed in this section considers cumulative temporal aggregates.

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* [PS85] and the *B-tree* [BM72]. 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.

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 7) 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

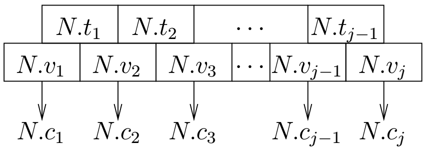

Figure 7: An interior node N.

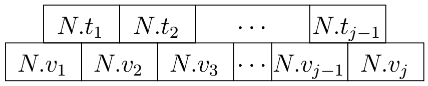

Figure 8: A leaf node N.

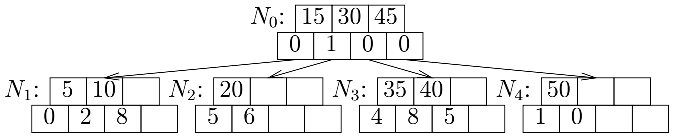

Figure 9: SB-tree for $SumDosage$.

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 8). 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 9 shows an SB-tree index for the aggregate $SumDosage$ from Figure 3 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 as follows:

- 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 of $N.I_i$, denoted $end(N.I_i)$, 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 specified as follows:

- $\big[start(N.I_i), end(N.I_i)\big)$, if $start(N.I_i) \neq -\infty$.
- $\big(-\infty, end(N.I_i)\big)$, if $start(N.I_i) = -\infty$.

For example, in Figure 9, the first interval of node N_0 is $(-\infty, 15)$, the second interval of node N_1 is $[5, 10)$, the last interval of node N_3 is $[40, 45)$, and the last interval of node 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 aggregate value 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 9. 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 of the lookup 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 Reconstruction of the Entire Aggregate

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., Figure 3, except I need not be the entire timeline.) 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 type of the aggregate:[1]

- For SUM and COUNT, $v_0 = 0$.
- For AVG, $v_0 = \langle 0, 0\rangle$.
- For MIN and MAX, $v_0 = \texttt{NULL}$.

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 9, $range(N_0, [14, 28), 0)$ 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 Figure 3.

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 9, $range(N_0, (-\infty, \infty), 0)$ returns the contents of the temporal aggregate $SumDosage$ as shown in Figure 3.

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, range queries on SB-trees have the same asymptotic running time as range queries on B-trees.

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, $v = v_{base}$.

[1] It is also acceptable to define $v_0 = \texttt{NULL}$ 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.

- 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 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 Figure 1, only $N_0.v_1$ and $N_0.v_2$ in Figure 9 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 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 9 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 9. 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 10.

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

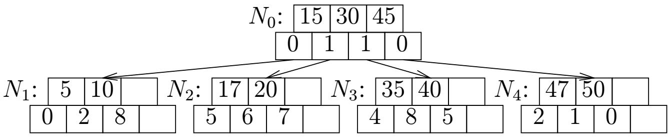

Figure 10: SB-tree after *insert*.

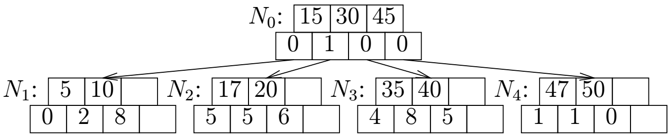

Figure 11: SB-tree after *delete*.

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 the tuple $\langle$"Ida"$, 1, [17, 47)\rangle$ that 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 10, we obtain the SB-tree in Figure 11. Notice that Figure 11 is not identical to Figure 9, the SB-tree before the insertion and the deletion. In particular, the first and the second intervals of N_2 in Figure 11 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 as a way of compacting 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.

Formally, suppose that an overflowing node N currently contains n intervals, where $n = l + 1$ or $l + 2$ if N is a leaf, or $n = b + 1$ if N is not a leaf. In the following, we define the procedure *split*(N), which reorganizes the SB-tree to deal with the overflow at node N.

- Split N into N_1 and N_2, such that:

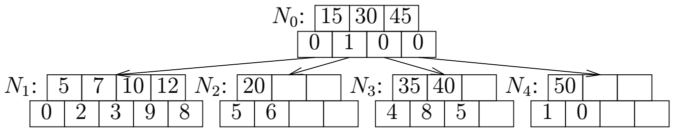

Figure 12: SB-tree before $split(N_1)$.

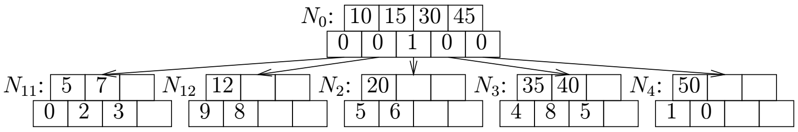

Figure 13: SB-tree before $split(N_0)$.

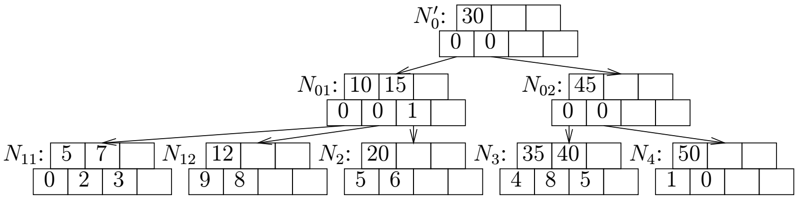

Figure 14: SB-tree after *split* completes.

  - N_1 contains the first $\lceil \frac{n}{2} \rceil$ intervals of N; that is, N_1 contains time instants $N.t_1, \ldots, N.t_{\lceil \frac{n}{2} \rceil - 1}$, aggregate values $N.v_1, \ldots, N.v_{\lceil \frac{n}{2} \rceil}$, and if N is not a leaf, child pointers $N.c_1, \ldots, N.c_{\lceil \frac{n}{2} \rceil}$.
  - N_2 contains the remaining intervals of N; that is, N_2 contains time instants $N.t_{\lceil \frac{n}{2} \rceil + 1}, \ldots, N.t_{n-1}$, aggregate values $N.v_{\lceil \frac{n}{2} \rceil + 1}, \ldots, N.v_n$, and if N is not a leaf, child pointers $N.c_{\lceil \frac{n}{2} \rceil + 1}, \ldots, N.c_n$.
- If N is the root node, create a new root node N' with two intervals that point to N_1 and N_2. Set $N'.t_1 = N.t_{\lceil \frac{n}{2} \rceil}$, $N'.c_1 = N_1$, $N'.c_2 = N_2$, and $N'.v_1 = N'.v_2 = v_0$, where v_0 is the aggregate-dependent value defined in Section 3.2. (Recall from the beginning of Section 3 that it is permissible for the root to have only two intervals.)
- If N is not the root node, then suppose N has a parent node N' with $N'.c_j = N$.
  - Split the j-th interval of N' into two intervals and have them point to N_1 and N_2. Specifically:
    - The first $j - 1$ intervals of N' stay the same; that is, $N'.t_i$, $N'.c_i$, and $N'.v_i$ remain unchanged for all $i < j$.
    - Starting from the $(j+1)$-st, each interval is moved one position to the right; that is, $N'.t_{i-1}$ becomes $N'.t_i$, $N'.c_i$ becomes $N'.c_{i+1}$, and $N'.v_i$ becomes $N'.v_{i+1}$, for all $i > j$.
    - Set $N'.t_j = N.t_{\lceil \frac{n}{2} \rceil}$, $N'.c_j = N_1$, $N'.c_{j+1} = N_2$, and $N'.v_{j+1} = N'.v_j$. $N'.v_j$ remains unchanged.
  - If N' overflows, call $split(N')$.

For example, consider executing $insert(N_0, \langle 1, [7, 12) \rangle)$ on the SB-tree in Figure 9. The resulting SB-tree before any node splitting is shown in Figure 12. 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 13. 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 the SB-tree shown in Figure 14.

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 the depth of the recursion in *split* is limited by the depth of the SB-tree, 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 $O(h)$.

3.6 Interval and Node Merging

At this point, it might appear that we have provided 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 11). 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 11. 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.

We now describe the interval merging procedure *imerge* in detail. It relies on a node merging procedure *nmerge* to be described shortly. In general, two adjacent intervals in leaves should be merged if they have the same aggregate value. An important case to note is when the adjacent intervals are in the same leaf. Since the intervals are on the same path from the root, they have the same aggregate value if and only if they have the same leaf value. Thus there are two cases:

- The two adjacent intervals belong to the same leaf N. Suppose that they are $N.I_j$ and $N.I_{j+1}$. If $N.v_j = N.v_{j+1}$, then:
  - Merge $N.I_j$ and $N.I_{j+1}$ into one interval by removing $N.t_j$ and $N.v_{j+1}$ from N. Specifically:
    - ⋄ For all $i < j$, $N.t_i$ and $N.v_i$ remain unchanged.
    - ⋄ For all $i > j + 1$, replace $N.t_{i-2}$ with $N.t_{i-1}$ and $N.v_{i-1}$ with $N.v_i$.
  - If N now contains fewer than $\lceil \frac{l}{2} \rceil$ intervals, call $nmerge(N)$.
- The two adjacent intervals belong to two different leaves N_1 and N_2. Suppose the intervals are $N_1.I_j$, the last interval of N_1, and $N_2.I_1$, the first interval of N_2. Let N be the least common ancestor of N_1 and N_2. Suppose N_1 is in the subtree rooted at $N.c_k$ and N_2 is in the subtree rooted at $N.c_{k+1}$. If $lookup(N.c_k, start(N_1.I_j)) = lookup(N.c_{k+1}, start(N_2.I_1))$ (note that $N_1.v_j$ and $N_2.v_1$ could be different), then:
  - If N_1 contains more than $\lceil \frac{l}{2} \rceil$ intervals, merge $N_1.I_j$ into $N_2.I_1$. Specifically:
    - ⋄ In N, set $N.t_k = N_1.t_{j-1}$.
    - ⋄ In N_1, remove $N_1.t_{j-1}$ and $N_1.v_j$.
  - Otherwise, merge $N_2.I_1$ into $N_1.I_j$. Specifically:
    - ⋄ In N, set $N.t_k = N_2.t_1$.
    - ⋄ In N_2, remove $N_2.t_1$ and $N_2.v_1$. Then, for all $i > 1$, replace $N_2.t_i$ with $N_2.t_{i+1}$ and $N_2.v_i$ with $N_2.v_{i+1}$.
    - ⋄ If N_2 now contains fewer than $\lceil \frac{l}{2} \rceil$ intervals, call $nmerge(N_2)$.

The *nmerge* procedure is used by *imerge* to fix underfull nodes. 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

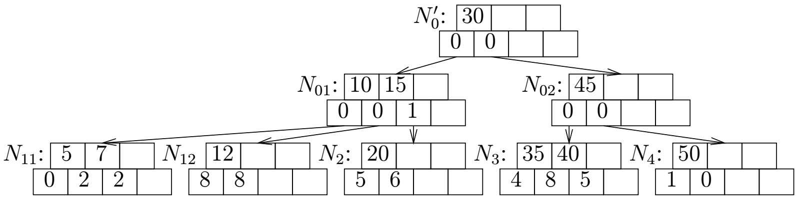

Figure 15: SB-tree before $imerge$.

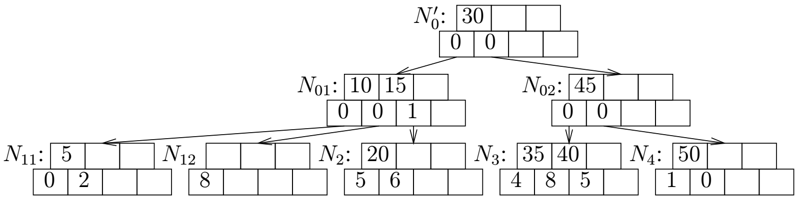

Figure 16: SB-tree before $nmerge(N_{1_2})$.

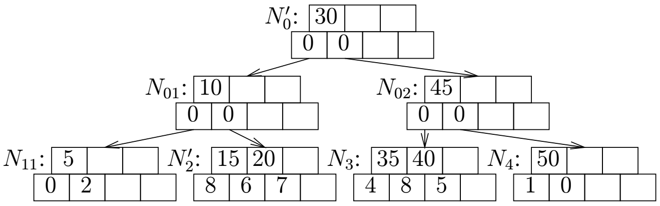

Figure 17: SB-tree after $imerge$ completes.

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. Although this high-level description of $nmerge$ is short, the details are quite involved because we must manipulate 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. The procedure $nmerge(N)$ is specified in full in Appendix A.

As a simple example, consider the SB-tree in Figure 11. We run $imerge$ twice, first on the 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 9. In this example, $nmerge$ is not needed.

As a more complicated example, let us continue with the example of 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 14; the result is shown in Figure 15. 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 16 shows the state of the SB-tree right before $imerge$ calls $nmerge(N_{12})$ 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 17. Notice that the SB-tree in Figure 17 is not identical to the one we started with in Figure 9. Nevertheless, they encode exactly the same aggregate.

As an exhaustive example, Figure 24 in Appendix B shows the sequence of snapshots of the SB-tree for aggregate $SumDosage$ as tuples are inserted into $Prescription$ in the order listed in Figure 1 and then deleted in the reverse order. The first and the last snapshots in Figure 24 are both empty SB-trees. 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

After an insertion or deletion, *imerge* should be called for each pair of adjacent leaf intervals with equal aggregate values in the SB-tree. First, we must be able 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 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.

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* 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 *imerge* 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 SB-tree update 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 aggregate result.

For `MIN` and `MAX`, each SB-tree update, including calls to *insert* and *split* but not *imerge*, 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. As an optimization, we periodically compact the SB-tree using *bmerge*, whose running time is $O(n + m \log m)$.

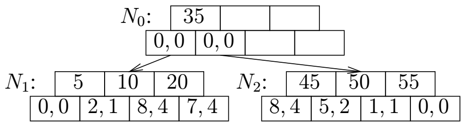

Figure 18: SB-tree for $AvgDosage_5$.

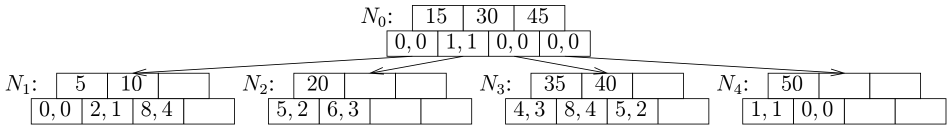

Figure 19: SB-tree for $AvgDosage$.

4 Cumulative Temporal Aggregates

In this section we show how to use SB-trees to compute and maintain cumulative temporal aggregates. Recall from Section 1 that a cumulative temporal aggregate is computed with an additional parameter w, for window offset. The value of the cumulative aggregate at time instant t is calculated over all base tuples that are valid at some point during the window $[t - w, t]$. None of the related work discussed in Section 2 addresses the problem of incrementally computing and maintaining cumulative temporal aggregates. We divide the problem into two cases and tackle them separately. First, we consider the case where a fixed window offset is known in advance. Then we show how to support cumulative aggregates with arbitrary window offsets. The latter case requires two solutions: one for `SUM`, `COUNT`, and `AVG` aggregates, and another for `MIN` and `MAX` aggregates.

4.1 Cumulative Aggregates with Fixed Window Offsets

We use one SB-tree to support each cumulative aggregate whose window offset w is known in advance. The heart of the problem is to determine the correct effect of a base update on the cumulative aggregate. As shown in Section 3.3, we can capture the effect of an insertion into a base table on an instantaneous aggregate by a pair $\langle v, I\rangle$, where $I = [I_{start}, I_{end})$ is the valid interval of the inserted tuple. It turns out that the effect of this insertion on a cumulative aggregate with window offset w can be captured by a pair $\langle v, [I_{start}, I_{end} + w)\rangle$. For any time instant $t \in [I_{start}, I_{end} + w)$, the inserted tuple is valid during $[t - w, t]$ and hence contributes to the aggregate value at t. Therefore, to process this insertion, we call $insert(N, \langle v, [I_{start}, I_{end} + w)\rangle)$ on the SB-tree rooted at N. If necessary, we then call *split* and/or *imerge* as discussed in Sections 3.5 and 3.6. Deletions are handled analogously. Procedures *lookup* and *range* require no change. All operations have the same asymptotic running times as their counterparts for instantaneous aggregates.

As an example, Figure 18 shows the SB-tree index for the cumulative `AVG` aggregate $AvgDosage_5$ with window offset 5, whose contents are shown in Figure 5. Looking up the value of $AvgDosage_5$ at time instant 32 in this SB-tree, we get $accum(N_0.v_1, N_1.v_4) = accum(\langle 0, 0\rangle, \langle 7, 4\rangle) = \langle 7, 4\rangle$. Therefore, the `AVG` value is $7/4 = 1.75$. For the purpose of comparison, Figure 19 shows the SB-tree index for the instantaneous `AVG` aggregate $AvgDosage$, which can be thought of as a cumulative aggregate with window offset 0. The contents of $AvgDosage$ are shown in Figure 4. The value of $AvgDosage$ at time 32 is $4/3 = 1.33$.

The solution above is quite straightforward and can be used in many data warehousing scenarios where the warehouse users are mostly interested in cumulative temporal aggregates with a few popular window

R_1:

value	*valid*
1	$[10, 20)$
1	$[20, 30)$

R_2:

value	*valid*
1	$[10, 30)$

Instantaneous `SUM` over R_1 or R_2:

value	*valid*
1	$[10, 30)$

Cumulative `SUM` over R_1 ($w = 10$):

value	*valid*
1	$[10, 20)$
2	$[20, 30)$
1	$[30, 40)$

Cumulative `SUM` over R_2 ($w = 10$):

value	*valid*
1	$[10, 40)$

Figure 20: A cumulative `SUM` aggregate is not computable from an instantaneous `SUM` aggregate.

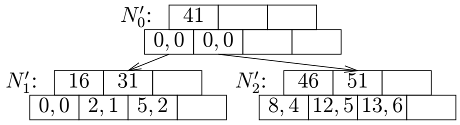

Figure 21: SB-tree T' for cumulative `AVG` aggregates over *Prescription* with arbitrary window offsets.

offsets such as a day, a week, or thirty days. We need one SB-tree index for each cumulative aggregate for each window offset, because in general an SB-tree index intended for a specific window offset cannot be used for a different window offset.

4.2 Cumulative `SUM`, `COUNT`, and `AVG` Aggregates with Arbitrary Window Offsets

For `SUM`, `COUNT`, and `AVG`, it is not always possible to compute a cumulative aggregate from the SB-tree index for the corresponding instantaneous aggregate. An example is shown in Figure 20. The instantaneous `SUM` aggregates over two two base tables R_1 and R_2 have the same contents. On the other hand, the cumulative `SUM` aggregates over these two tables using window offset $w = 10$ are different. Looking at the instantaneous aggregate alone, we cannot tell whether it is computed over R_1 or R_2; therefore, we have no way of knowing the correct result for the cumulative aggregate.

Our solution is to maintain another 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. For example, in order to support cumulative `AVG` aggregates over *Prescription* with arbitrary window offsets, we maintain the SB-tree T shown in Figure 19 as well as the SB-tree T' shown in Figure 21. The various operations on T and T' are described next.

Lookup The value of the cumulative aggregate with window offset w at time instant t is calculated by:

$$accum\big(lookup(N, t), diff(lookup(N', t), lookup(N', t - w))\big).$$

Here, *diff* is a function that computes the difference between two aggregate values according to the type of the aggregate. The definition of *diff* follows. Again, recall that we treat an `AVG` aggregate value as a pair of `SUM` and `COUNT` values.

- For `SUM` and `COUNT`, $diff(x, y) = x - y$.
- For `AVG`, $diff(\langle x_{sum}, x_{count}\rangle, \langle y_{sum}, y_{count}\rangle) = \langle x_{sum} - y_{sum}, x_{count} - y_{count}\rangle$.

Intuitively, $diff(lookup(N', t), lookup(N', t - w))$ returns an aggregate value that is computed over all tuples whose valid intervals do not contain t but overlap with $[t - w, t]$. We then add $lookup(N, t)$, which returns an aggregate value computed over all tuples whose valid intervals contain t and hence overlap with $[t-w, t]$. The result is an aggregate value computed over all tuples whose valid intervals overlap with $[t-w, t]$, i.e., the value of the cumulative aggregate at time t with window offset w.

As an example, let us calculate the value of the cumulative aggregate $AvgDosage_5$ (which has an window offset of 5) at time instant 19 using this method. For the SB-tree T' in Figure 21, $\mathit{lookup}(N', 19) = \mathit{accum}(\langle 0,0\rangle, \langle 2,1\rangle) = \langle 2,1\rangle$, and $\mathit{lookup}(N', 19-5) = \mathit{accum}(\langle 0,0\rangle, \langle 0,0\rangle) = \langle 0,0\rangle$. For the SB-tree T in Figure 19, $\mathit{lookup}(N, 19) = \mathit{accum}(\langle 1,1\rangle, \langle 5,2\rangle) = \langle 6,3\rangle$. Therefore, the value of $AvgDosage_5$ at time 19 is $\mathit{accum}\big(\langle 6,3\rangle, \mathit{diff}\,(\langle 2,1\rangle, \langle 0,0\rangle)\big) = \langle 8,4\rangle$, which is consistent with the answer we obtained in Section 4.1 from the SB-tree built specifically for $AvgDosage_5$ in Figure 18.

Range Query A range query over the interval $I = [I_{start}, I_{end})$ can be answered by combining the results of two range queries $\mathit{range}(N', [I_{start} - w, I_{end}), v_0)$ and $\mathit{range}(N, I, v_0)$, where v_0 is the aggregate-dependent value defined in Section 3.2. We can avoid producing the intermediate results by coordinating the DFT's on T and T'. The detailed procedure is quite involved but not difficult to grasp. It is omitted from this paper due to space constraints.

Insertion, Deletion, Splitting, and Merging When there is an insertion into the base table, we maintain T as discussed in Section 3. We then maintain T' as follows. Suppose that the effect of this insertion on T is the pair $\langle v, I\rangle$, where $I = [I_{start}, I_{end})$ is the valid interval of the inserted tuple. Then, the effect of this insertion on T' should be the pair $\langle v, (I_{end}, \infty)\rangle$, since $\mathit{lookup}(N', t)$ is supposed to return an aggregate value computed over all tuples that are valid strictly before t. To process this insertion, we call $insert(N', \langle v, (I_{end}, \infty)\rangle)$. If necessary, we then call *split* and/or *imerge* as in Section 3. Deletions are handled in analogously.

Discussion In summary, the solution presented above handles cumulative `SUM`, `COUNT`, and `AVG` aggregates with arbitrary window offsets. Since it uses two SB-trees, all operations take two to three times as long as their counterparts for instantaneous aggregates. Nevertheless, their asymptotic running times are the same.

4.3 Cumulative `MIN` and `MAX` Aggregates with Arbitrary Window Offsets

Unlike the `SUM`, `COUNT`, and `AVG` aggregates discussed in the previous section, it is possible to compute a cumulative `MIN` or `MAX` aggregate with arbitrary window offset from the SB-tree index constructed for the corresponding instantaneous aggregate. Suppose we have an SB-tree T (rooted at N) for an instantaneous `MIN` or `MAX` aggregate. To find the value of the cumulative aggregate with window offset w at time instant t, we simply call $\mathit{range}(N, [t-w, t], v_0)$, 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. Recall from Section 3.2 that the running time of *range* is $O(h+r)$, where h is the height of the SB-tree and r is the number of leaves that intersect with $[t-w, t]$. This running time may be too slow for a lookup operation when w is large.

We can reduce the running time of *lookup* to $O(h)$ by storing additional information inside non-leaf nodes. For each interval $N.I_i$ in a non-leaf node N, we store a "u" value denoted $N.u_i$ in addition to the "v" value $N.v_i$. We call this new new index structure an *MSB-tree* (for `MIN`/`MAX` SB-tree).

To describe the property of MSB-trees, we use `MAX` aggregates as an example; the case of `MIN` aggregates is analogous. Suppose that N is a not a leaf. Recall from Section 3.1 that if we compute the maximum of all v values along the path from the root to $N.I_i$ including $N.v_i$, we will obtain a lower bound for the `MAX` value

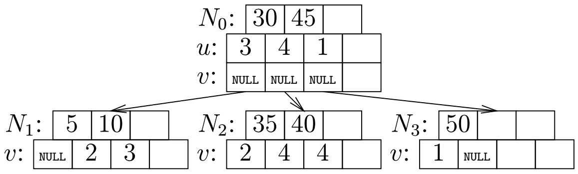

Figure 22: MSB-tree for cumulative MAX aggregates over *Prescription* with arbitrary window offsets.

during every leaf interval in the subtree rooted at $N.c_i$. In effect, this value also serves as a lower bound for the MAX value during $N.I_i$. Now, with help from $N.u_i$, we can determine the exact MAX value during $N.I_i$: it is the bigger one of $N.u_i$ and the lower bound computed from the v values. There is no need to store any u value for a leaf interval, because the lower bound computed from the v values is in fact the exact MAX value during the leaf interval.

For example, Figure 22 shows an MSB-tree that supports cumulative MAX aggregates over *Prescription* with arbitrary window offsets. This MSB-tree has not been compacted yet, i.e., there are adjacent leaf intervals with equal MAX values. Next, we discuss in detail the operations on an MSB-tree, including a batch operation that compacts the tree.

Lookup The MSB-tree lookup function $mlookup(N, t, w, u)$ searches the MSB-tree rooted at node N and returns the value of the cumulative aggregate with window offset w at time instant t. The fourth parameter u is used to pass partially calculated aggregate values to recursive calls. Initially, we start the *mlookup* call with $u = v_0$, where v_0 is the value defined in Section 3.2. The definition of $mlookup(N, t, w, u)$ follows.

- Let $u_{my} = u$.
- For each i such that $N.I_i \cap [t - w, t] \neq \varnothing$:
  - If N is a leaf, set $u_{my} = accum(u_{my}, N.v_i)$.
  - Otherwise, N is not a leaf:
    - ◇ If $u_{my} = accum(accum(u_{my}, N.u_i), N.v_i)$, do nothing.
    - ◇ Otherwise, if $N.I_i \subseteq [t - w, t]$, set $u_{my} = accum(accum(u_{my}, N.u_i), N.v_i)$.
    - ◇ Otherwise, set $u_{my} = mlookup(N.c_i, t, w, accum(u_{my}, N.v_i))$.
- Return u_{my}.

The local variable u_{my} holds the MIN or MAX value seen by *mlookup* so far. We use the u and v values whenever possible in order to avoid recursing down to subtrees. For example, Figure 22 shows an MSB-tree for cumulative MAX aggregates over *Prescription*. Let us look up the value of $MaxDosage_{20}$ (with window offset 20) at time 50. At the root, the first interval that overlaps with $[50 - 20, 50] = [30, 51)$ is $N_0.I_2 = [30, 45)$. Since $N_0.I_2 \subseteq [30, 51)$, we get a MAX value of 4, and there is no need to recurse down to N_2. Next, we move on to $N_0.I_3 = [45, \infty)$, which also overlaps with $[30, 51)$. The MAX value during the entire $N_0.I_3$ is 1, less than the MAX value of 4 that we have obtained so far. Therefore, there is no need to recurse down to N_3 either. Hence, the value at time 50 is 4, which is consistent with the contents of $MaxDosage_{20}$ shown in Figure 6.

Notice that this procedure is almost identical in structure to the SB-tree *insert* procedure defined in Section 3.3. Therefore, the running time of *mlookup* is also $O(h)$, where h is the height of the MSB-tree.

Range Query A range query over an interval $I = [I_{start}, I_{end})$ can be answered easily by a pass over the result of $range(N, [I_{start} - w, I_{end}), v_0)$, where N is the root of the MSB-tree and v_0 is the value defined in Section 3.2. However, $range$ does not take advantage of the u values stored in non-leaf nodes. As an optimization, we can revise the procedure to make use of the u values and avoid producing any intermediate result. The details are rather complicated and are omitted due to space constraints.

Insertion, Splitting, and Merging Suppose that an insertion into the base table has an effect of $\langle v, I\rangle$ on the instantaneous aggregate. To update the MSB-tree, we use the procedure $minsert(N, \langle v, I\rangle)$, where N is the root of the MSB-tree. The definition of $minsert$ follows.

- For each i such that $N.I_i \cap I \neq \varnothing$:
  - If N is not a leaf, set $N.u_i = accum(v, N.u_i)$.
  - 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, $N.I_i \nsubseteq I$.
    - 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 $\langle v, I\rangle$.

This procedure is identical to the SB-tree $insert$ procedure defined in Section 3.3 except the additional line at the beginning of the loop. Intuitively, we update the u value for any non-leaf interval $N.I_i$ that overlaps with I, even if $N.I_i$ is not completely contained in I, because the u value tries to record the `MIN` or `MAX` value during the entire $N.I_i$. On the other hand, we cannot update the v value for $N.I_i$ if $N.I_i$ is not completely contained in I, because there may be leaf intervals in the subtree rooted at $N.c_i$ that are not affected by the insertion. Clearly, $minsert$ has the same asymptotic running time of $O(h)$ as $insert$, where h is the height of the MSB-tree.

When a node N becomes overfull as the result of $minsert$, we split it by calling $msplit(N)$. The $msplit$ procedure is identical to the SB-tree $split$ procedure defined in Section 3.5, except that we also need to preserve the u values when we split N, and set the the two u values in N's parent. Suppose that N is split into N_1 and N_2. The u value for N_1's parent interval is calculated by aggregating all the u and v values in N_1; similarly, the u value for N_2's parent interval is calculated by aggregating all the u and v values in N_2. Clearly, $msplit$ has the same asymptotic running time of $O(h)$ as $split$, where h is the height of the MSB-tree.

We do not perform interval and node merging after every insertion. Instead, we periodically compact the MSB-tree with a batch procedure $mbmerge$ similar to the SB-tree $bmerge$ procedure discussed in Section 3.6. The analysis of running time is similar to the SB-tree case covered at the end of Section 3.6. Also, as discussed in Section 3.4, we do not handle deletions for `MIN` and `MAX` aggregates.

As a complete example, Figure 25 in Appendix B shows the sequence of snapshots of the MSB-tree for cumulative `MAX` aggregates over *Prescription* as tuples are inserted into *Prescription* in the order listed in Figure 1. The last snapshot shows the result of running $mbmerge$ on this MSB-tree.

Discussion In summary, the solution presented above handles cumulative `MIN` and `MAX` aggregates with arbitrary window offsets. Since an MSB-tree stores more information in its non-leaf nodes than an SB-tree, an MSB-tree has a smaller maximum branching factor and hence more levels than an SB-tree with the same node size and the same number of leaf intervals. Therefore, the MSB-tree operations are a constant factor slower than their SB-tree counterparts.

5 Endpoint Sort Algorithm for Computing Temporal SUM, COUNT, and AVG

In Section 6 we will provide a comparison chart of all of the temporal aggregation algorithms discussed in Section 2 and the SB-tree algorithms presented in this paper, considering features such as running time for various operations, whether incremental updates are supported, and whether the algorithms are memory-based on disk-based. For completeness, we quickly present a disk-based version of the algorithm in [MLI00] for instantaneous aggregates SUM, COUNT, and AVG. The [MLI00] algorithm uses a memory-based balanced tree structure whose main purpose is for sorting. Here we present a more general *endpoint sort algorithm* that does not require any specific data structure. Instead, it can exploit any disk-based sorting algorithm supported by the database system.

We are given a base table R, and we wish to compute an instantaneous temporal SUM, COUNT, or AVG aggregate over a column of R. The algorithm follows.

- Consider each tuple of R in turn. Suppose that the effect of this tuple on the aggregate is $\langle v, I\rangle$, where $I = [I_{start}, I_{end})$ is the valid interval of this tuple (Section 3.3). Generate two tuples $\langle v, I_{start}\rangle$ and $\langle \mathit{diff}(v_0, v), I_{end}\rangle$, where v_0 and diff are as defined in Sections 3.2 and 4.2, respectively.
- Sort all generated tuples by their second attribute in ascending order. If two tuples $\langle v_1, t\rangle$ and $\langle v_2, t\rangle$ have the same value t for the second attribute, replace them with one tuple $\langle \mathit{accum}(v_1, v_2), t\rangle$ (or remove them if $\mathit{accum}(v_1, v_2) = v_0$), where accum is as defined in Section 3.1. Note that this step can be combined with the first one.
- Set $t_{my} = -\infty$ and $v_{my} = v_0$.
- For each generated tuple $\langle v, t\rangle$ in the sorted order:
  - Output $\langle v, [t_{my}, t)\rangle$ (or $\langle v, (t_{my}, t)\rangle$ if $t_{my} = -\infty$).
  - Set $t_{my} = t$ and $v_{my} = \mathit{accum}(v_{my}, v)$.

The intuition behind the endpoint sort algorithm is as follows. For each tuple, we mark the beginning and the end of its valid interval with its "positive" and "negative" effects on the aggregate value, respectively. We then sort the beginning and the end points of all valid intervals together. As an example, for base table *Prescription* in Figure 1, the first three tuples generated by the algorithm after the sorting step are $\langle 2, 5\rangle$, $\langle 6, 10\rangle$, and $\langle -2, 15\rangle$. The first tuple, $\langle 2, 5\rangle$, reflects the fact that Dan's prescription starts at time 5. The second tuple, $\langle 6, 10\rangle$, reflects that the prescriptions for Amy, Ben, and Fay all start at time 10. The third tuple, $\langle -2, 15\rangle$, reflects that Dan's prescription stops at time 15. Finally, in the last step of algorithm, we travel along the time line and update the running aggregate value whenever we encounter any beginning or end points.

6 Conclusion

We have presented a new index structure for temporal aggregation called the SB-tree. SB-trees provide many significant improvements over the previous approaches to implementing temporal aggregates. Like B-trees, SB-trees are balanced, disk-based index structures with good performance guarantees; they are well suited for computing and maintaining temporal aggregates over large quantities of temporal data. By incorporating

	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 [Tum92]	all	disk	$O(n^2)$	no	no	no
balanced tree [MLI00]	`SUM/COUNT/AVG`	memory	$O(n \log n)$	no	no	no
endpoint sort (Section 5)	`SUM/COUNT/AVG`	disk	$O(n \log n)$	no	no	no
merge sort [MLI00]	`MIN/MAX`	disk	$O(n \log n)$	no	no	no
aggregation tree [KS95]	all	memory	$O(n^2)$	$O(n)$	$O(n)$ (no if k-ordered)	no
SB-tree (Sections 3, 4.1)	all	disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	fixed window offset
dual SB-trees (Section 4.2)	`SUM/COUNT/AVG`	disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	arbitrary window offset
MSB-tree (Section 4.2)	`MIN/MAX`	disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	arbitrary window offset

Figure 23: Comparison of temporal aggregation algorithms (n is the size of the base table).

features from segment-trees, SB-trees are more efficient to maintain than materialized temporal aggregates, especially in the presence of base tuples with long valid intervals. Furthermore, SB-trees contain enough information to construct the contents of the temporal aggregates that they index. These features make SB-trees a particularly effective data structure for supporting temporal aggregates in data warehouses.

We have also described three approaches to handling cumulative temporal aggregates. The first approach requires only a slight change to the the SB-tree insertion procedure, and is applicable in the case where a cumulative aggregate has a fixed window offset known in advance. The second approach uses a pair of SB-trees to handle cumulative `SUM`, `COUNT`, and `AVG` aggregates. The third approach uses an extension to the SB-tree called the MSB-tree to handle cumulative `MIN` and `MAX` aggregates. Compared to the basic SB-tree, the last two approaches 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.

In Figure 23, we compare our SB-tree algorithms and the endpoint sort algorithm of Section 5 with the other temporal aggregation algorithms discussed in Section 2. For simplicity of presentation, Figure 23 provides only rough upper bounds on the running times of algorithms; please refer to the the appropriate sections of this paper for detailed analyses.

As future work, we plan to implement SB-trees and MSB-trees in a disk-based environment and measure their performance with real-world applications. We also plan to consider concurrency control algorithms for SB-trees and MSB-trees.

References

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[Eps79] R. Epstein. Techniques for processing of aggregates in relational database systems. Technical Report UCB/ERL M7918, University of California, Berkeley, California, 1979.

[GHR+99] 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.

[KS95] 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.

[MLI00] 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*, San Diego, California, March 2000. To appear.

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

[SGM93] 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.

[Sno95] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[Tum92] P. A. Tuma. Implementing historical aggregates in TempIS. Master's thesis, Wayne State University, Detroit, Michigan, 1992.

[YK97] 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.

[YW98] 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.

[YW00] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. In *Proc. of the 2000 Intl. Conf. on Extending Database Technology*, Konstanz, Germany, March 2000. To appear.

A Node Merging Procedure

Suppose that node N is less than half full. The node merging procedure $nmerge(N)$ is specified as follows:

- If N is the root:
  - If N has exactly one child, make $N.c_1$ the new root, set $N.c_1.v_i = accum(N.v_1, N.c_1.v_i)$ for all i, and then delete the old root N.
  - Otherwise, do nothing.
- Otherwise, N is not the root. Suppose that N can hold a maximum of n intervals ($n = l$ if N is a leaf; $n = b$ otherwise). Currently N contains only $\lceil \frac{n}{2} \rceil - 1$ intervals, one below the required minimum.
  - If N', the right sibling of N, contains at least $\lceil \frac{n}{2} \rceil + 1$ intervals, remove the first interval of N' and append it to N. Specifically:
    - ⋄ Suppose N_p is the parent of both N and N'. Moreover, $N_p.c_k = N$ and $N_p.c_{k+1} = N'$.
    - ⋄ In N, for all i, set $N.v_i = accum(N_p.v_k, N.v_i)$.
    - ⋄ In N_p, set $N_p.v_k = v_0$, where v_0 is the value defined in Section 3.2.
    - ⋄ In N, set $N.t_{\lceil \frac{n}{2} \rceil - 1} = N_p.t_k$, $N.v_{\lceil \frac{n}{2} \rceil} = accum(N_p.v_{k+1}, N'.v_1)$, and if N is not a leaf, $N.c_{\lceil \frac{n}{2} \rceil} = N'.c_1$.
    - ⋄ In N_p, set $N_p.t_k = N'.t_1$.
    - ⋄ In N', remove $N'.t_1$, $N'.v_1$, and if N' is not leaf, remove $N'.c_1$. For all $i > 1$, replace $N'.t_i$ with $N'.t_{i+1}$, $N'.v_i$ with $N'.v_{i+1}$, and if N' is not a leaf, $N'.c_i$ with $N'.c_{i+1}$.
  - Otherwise, if N', the left sibling of N, contains $j > \lceil \frac{n}{2} \rceil$ intervals, remove the last interval of N' and prepend it to N. Specifically:
    - ⋄ Suppose N_p is the parent of both N' and N. Moreover, $N_p.c_k = N'$ and $N_p.c_{k+1} = N$.
    - ⋄ In N, for all i, set $N.v_i = accum(N_p.v_{k+1}, N.v_i)$.
    - ⋄ In N_p, set $N_p.v_{k+1} = v_0$, where v_0 is the value defined in Section 3.2.
    - ⋄ In N, for all i, move $N.t_i$ to $N.t_{i+1}$, $N.v_t$ to $N.v_{t+1}$, and if N is not a leaf, $N.c_i$ to $N.c_{i+1}$. Then, set $N.t_1 = N_p.t_k$, $N.v_1 = accum(N_p.v_k, N'.v_j)$, and if N is not a leaf, $N.c_1 = N'.c_j$.
    - ⋄ In N_p, set $N_p.t_k = N'.t_{j-1}$.
    - ⋄ In N', remove $N'.t_{j-1}$, $N'.v_j$, and if N' is not a leaf, $N'.c_j$.
  - Otherwise, merge N with a sibling as follows. Let N_1 and N_2 denote these two nodes, from left to right. Suppose that N_1 contains j_1 intervals and N_2 contains j_2. One of j_1 and j_2 is $\lceil \frac{n}{2} \rceil$, while the other is $\lceil \frac{n}{2} \rceil - 1$.
    - ⋄ Suppose N_p is the parent of both N_1 and N_2. Moreover, $N_p.c_k = N_1$ and $N_p.c_{k+1} = N_2$.
    - ⋄ Merge N_1 and N_2 into a new node N' with $j_1 + j_2$ intervals (it is not difficult to verify that $\lceil \frac{n}{2} \rceil < j_1 + j_2 \leq n$). Specifically:
      - For $1 \leq i < j_1$, set $N'.t_i = N_1.t_i$.
        Set $N'.t_{j_1} = N_p.t_k$.
        For $j_1 < i < j_1 + j_2$, set $N'.t_i = N_2.t_{i-j_1}$.
      - For $1 \leq i \leq j_1$, set $N'.v_i = accum(N_p.v_k, N_1.v_i)$.
        For $j_1 < i \leq j_1 + j_2$, set $N'.v_i = accum(N_p.v_{k+1}, N_2.v_{i-j_1})$.

– If N_1 and N_2 are not leaves,
then for $1 \leq i \leq j_2$, set $N'.c_i = N_1.c_i$,
and for $j_1 < i \leq j_1 + j_2$, set $N'.c_i = N_2.c_{i-j_1}$.

– Delete the old nodes N_1 and N_2 (but not their descendents).

◇ In N_p, merge $N_p.I_k$ and $N_p.I_{k+1}$ into one interval and point it to N'. Specifically:

– Set $N_p.c_k = N'$ and $N_p.v_k = v_0$, where v_0 is the value defined in Section 3.2.

– Remove $N_p.t_k$ and $N_p.c_{k+1}$. Then, for all $i > k$, replace $N_p.t_{i-1}$ with $N_p.t_i$, $N_p.v_i$ with $N_p.v_{i+1}$, and $N_p.c_i$ with $N_p.c_{i+1}$.

◇ If N_p now contains fewer than $\lceil \frac{b}{2} \rceil$ intervals, call $nmerge(N_p)$.

B Comprehensive Examples

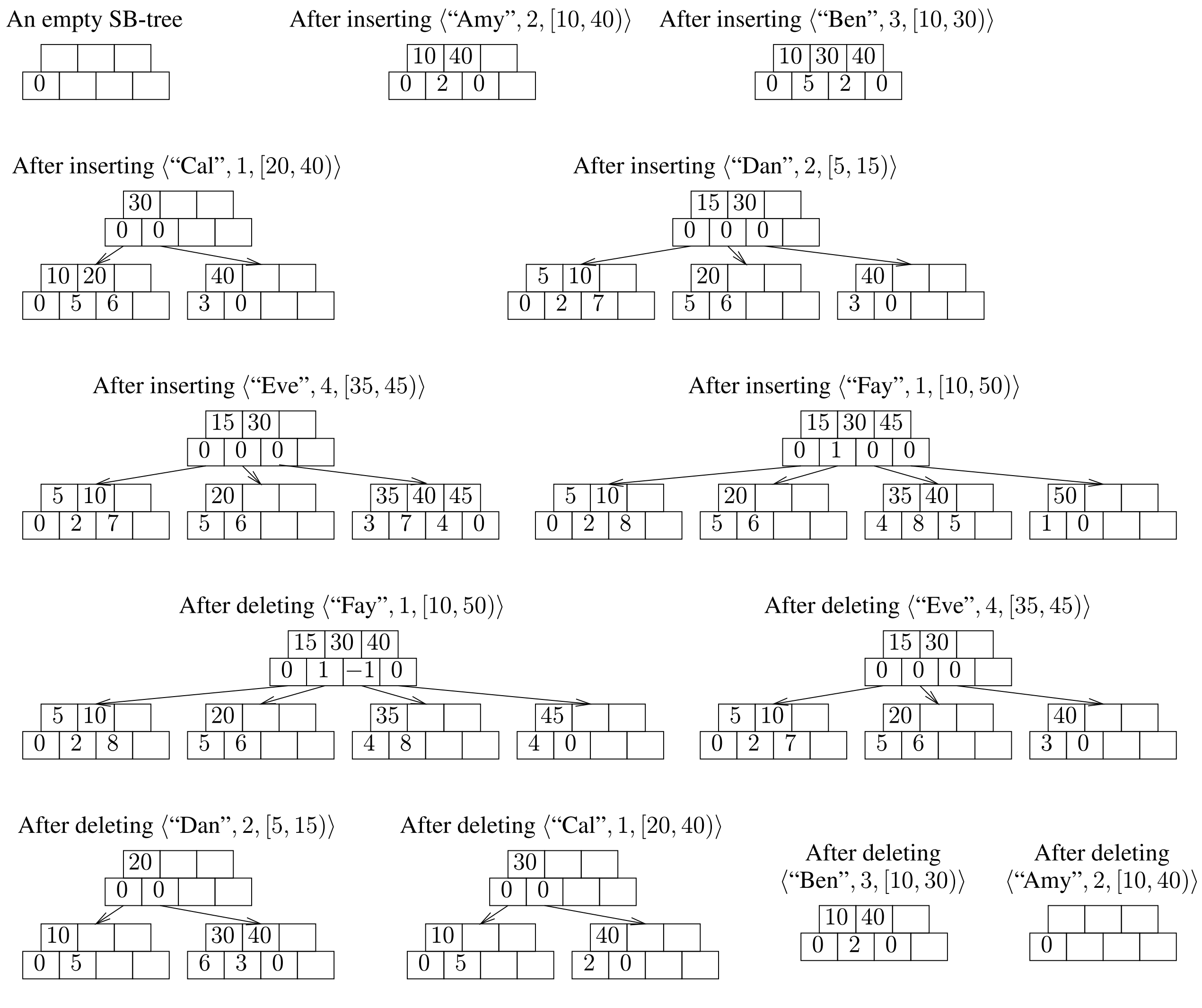

Figure 24: SB-tree for $SumDosage$.

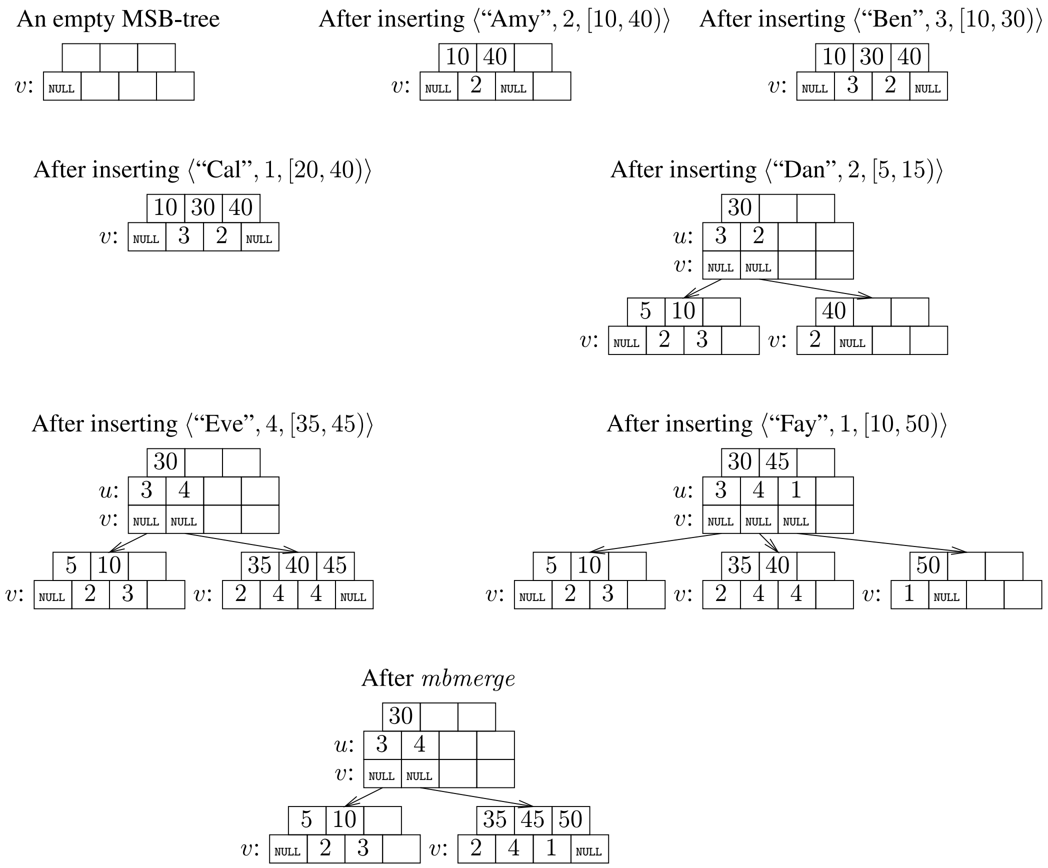

Figure 25: MSB-tree for cumulative `MAX` aggregates over *Prescription* with arbitrary window offsets.