The VLDB Journal (2003) 12: 262–283 / Digital Object Identifier (DOI) 10.1007/s00778-003-0107-z

Incremental computation and maintenance of temporal aggregates

Jun Yang[1], **Jennifer Widom**[2]

[1] Computer Science Department, Duke University (e-mail: junyang@cs.duke.edu)
[2] Computer Science Department, Stanford University (e-mail: widom@db.stanford.edu)

Edited by R. Snodgrass. Received: March 20, 2001 / Accepted: March 21, 2001
Published online: September 17, 2003 – © Springer-Verlag 2003

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 change. 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.

Keywords: Temporal database – Aggregation – View maintenance – Access methods – B-tree – Segment tree

1 Introduction

Temporal aggregation operators are included in most temporal query languages, including TQuel [19] and TSQL2 [18]. 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 (Sect. 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 nontemporal 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 1 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 Fig. 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 [19]. 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

This work was supported by the National Science Foundation under grant IIS-9811947 and by NASA Ames under grant NCC2-5278.

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	$[10, 50)$

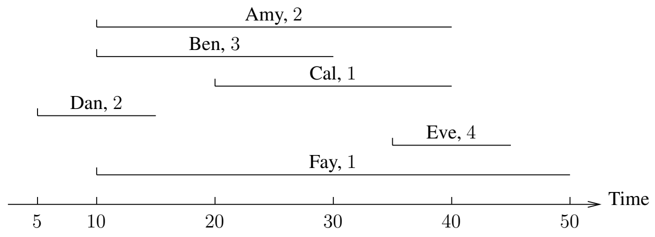

Fig. 1. Graphical representation of *Prescription*

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	$[5, 20)$
1.75	$[20, 30)$
1.67	$[30, 35)$
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	$[45, 50)$
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	$[70, \infty)$

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 [21]. 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 accumulate, 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 [22]. Another problem is that the traditional data warehousing approach of directly materializing and maintaining the view contents can be extremely inefficient for temporal aggregates. For 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 update *SumDosage* properly, 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 seventh tuples in Table 2. In other words, as a 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.

Even base table updates with short valid intervals can have significant effects on temporal aggregates. We present two more examples to support this point. The first example illustrates a very common case where the temporal aggregate is not defined directly over the base table but instead on a query over the base table. Consider an instantaneous temporal aggregate view *SumOldDosage*, which computes the sum of dosages over the "old" prescriptions, i.e., those that terminated before time 50. Initially, *SumOldDosage* is computed over all prescriptions in Table 1. Suppose we now extend the prescription for Fay by one day. This base table update has a very short interval $[50, 51)$. However, after this update, Fay's prescription period becomes $[10, 51)$ and is no longer considered "old." As a result, we must remove the contribution of Fay's prescription to *SumOldDosage* over the long interval $[10, 50)$. In the next example, we illustrate the effect of base table updates on cumulative temporal aggregates. Consider our earlier example view $MaxDosage_{20}$. Suppose we insert a new prescription record $\langle$"Hal"$, 6, [30, 31)\rangle$ into the base table. Again, this update has a very short valid interval, but it affects $MaxDosage_{20}$ over a much longer interval $[30, 51)$. The reason is that $MaxDosage_{20}$ is computed over all prescriptions active at any point within the past 20 days; therefore, an update at time 30 can affect the value of $MaxDosage_{20}$ up to time 50. In general, the length of the affected interval is determined by the window offset and can be arbitrarily large.

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 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 *dual SB-trees* to support cumulative `SUM`, `COUNT`, and `AVG` aggregates with arbitrary window offsets. We also consider an extension to the SB-tree called the *JSB-tree*, which provides interesting performance trade-offs with dual SB-trees. Finally, to support cumulative MIN and MAX aggregates with arbitrary window offsets, we introduce another extension to the SB-tree called the *MSB-tree*.

One of the major features of the SB-tree is its efficient handling of range updates, which greatly enhances the applicability of the SB-tree. Beyond the obvious application in view maintenance, SB-trees also can be used to compute temporal aggregates incrementally (by treating each input tuple as an insertion). Some previously proposed algorithms do not perform well if many input tuples have long valid intervals (Sect. 2), but SB-trees have no such problem.

SB-trees are useful to a wide spectrum of applications that need to maintain summary information over historical data. At one end of the spectrum we find traditional temporal database applications for which SB-trees are obviously suitable because the temporal database tables may undergo retroactive updates with long intervals. At the other end of the spectrum we find applications that work with append-only time-series data, including stock tickers, Web access logs, network traces, etc. For example, in a Web access log, updates have the form $\langle page, time \rangle$, and they usually arrive in increasing time order. At first glance, the range update feature of the SB-tree appears useless for this type of update. However, suppose we are interested in a summary of the access history for the set P of all recently accessed pages, i.e., pages that have been accessed within the past month. We define a cumulative temporal aggregate view V that computes, for each point in time, the total number of accesses to the pages in P during the last 24 h. Here, the window offset of 24 h controls the granularity of aggregation; an instantaneous aggregate would be too fine-grained because it counts the number of accesses at every second (the typical resolution of Web access logs). As discussed earlier, point updates can have long-range effects on temporal aggregates. For example, suppose a Web page p not currently in P is accessed at the current time. This append-only point update would require all past accesses to p to be counted in computing V. Furthermore, each access to p at time t, although a point update by itself, has a range effect on V for 24 h starting from t. Thus, for nontrivial temporal aggregate views such as V, it is still essential to support range updates efficiently.

Although introduced as a solution for computing and maintaining temporal aggregates, the SB-tree is capable of handling other types of aggregates. In general, the SB-tree and its variants can efficiently handle range data updates for both point-based and range-based aggregates (e.g., instantaneous and cumulative temporal aggregates, respectively) along any ordered key dimension. Work is already under way (e.g., [25]) to extend these index structures to handle multiple key and time dimensions.

2 Related work

Note that a significantly abridged version of this paper, omitting many details, results, and examples, appeared as [23].

A first proposal for computing temporal aggregates was given in [20] and was based on an extension to the nontemporal aggregate computation algorithm from [4]. 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.

Moon et al. [15] proposed a *balanced-tree algorithm* based on red-black trees for computing temporal SUM, COUNT, and AVG aggregates. In Sect. 5, we generalize the balanced-tree algorithm so that it is not tied to any particular data structure. We call this 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 [12] developed a structure called the *aggregation tree* based on the binary *segment-tree* [16]. 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 Sect. 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, 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 [24,6], but they all inherit the same limitations of the sequential version discussed above.

Much work has been done on indexing temporal data [17, 14]. Some temporal index structures use segment-trees. For example, Kolovson and Stonebraker [13] proposed the *SR-tree*, which combines the properties of the segment-tree and the *R-tree* [8]. However, segment-trees had never been used to index and maintain temporal aggregates until our proposal in [23]. Recent work by Zhang et al. [25] extended the SB-tree by combining it with the multiversion B-tree [2] to handle arbitrary key-range temporal SUM, COUNT, and AVG queries.

None of the related work discussed above considers cumulative temporal aggregates. On the other hand, the dual SB-tree technique we use to handle cumulative temporal SUM, COUNT, and AVG aggregates (Sect. 4.2) is quite reminiscent of the *prefix-sum* approach taken by Ho et al. [10] 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. [5] proposed the *dynamic data cube*. The two-dimensional case of the dynamic data cube, called the *cumulative B-tree*, has performance characteristics similar to those of 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 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 is an example of an *augmented data structure* [3]. It combines features from both the *segment-tree* [16] 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 require 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 that 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:

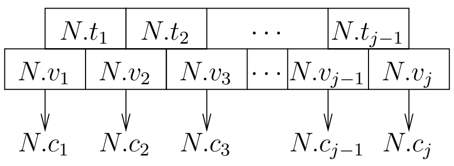

Fig. 2. An interior node N

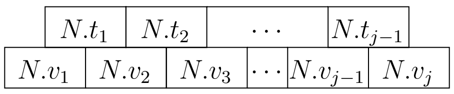

Fig. 3. A leaf node N

- 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 (Fig. 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 (Fig. 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 nonleaf 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, Fig. 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

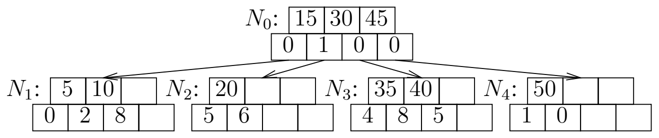

Fig. 4. SB-tree for *SumDosage*

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 Fig. 4, 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 several useful properties of SB-trees that are preserved by all SB-tree operations. (1) The time intervals in every node are contiguous, nonoverlapping, and sorted. (2) For any nonleaf node N, the i-th time interval $N.I_i$ is always the union of all time intervals in $N.c_i$. (3) 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. (4) All intervals' endpoints stored in an SB-tree are distinct, and they can be found in the base table. (5) The temporal aggregate value at time instant t can be computed by aggregating the partial aggregate values associated with all SB-tree intervals containing t. The last property ensures the correctness of SB-trees for computing temporal aggregates.

The rest of this section discusses SB-tree lookup (Sect. 3.1), range query (Sect. 3.2), insertion (Sect. 3.3), deletion (Sect. 3.4), reorganization (Sects. 3.5 and 3.6), bulk operations (Sect. 3.7), and, finally, a detailed performance analysis (Sect. 3.8).

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 t. Recall Property (5) of the SB-tree: the temporal aggregate value at t can be computed by aggregating the partial aggregate values associated with all SB-tree intervals containing t. We search the SB-tree recursively for these intervals, 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 $lookup(N, t) = N.v_i$.
- Otherwise $lookup(N, t) = N.v_i \oplus lookup(N.c_i, t)$.

In the above, $\oplus$ is a commutative operator that combines two aggregate values according to the type of the aggregate. The definition of $\oplus$ is shown below. Recall that we treat an AVG aggregate value as a pair of SUM and COUNT values.

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

For example, let us look up the value of the temporal aggregate *SumDosage* at time instant 19 using the SB-tree in Fig. 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 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 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 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 $N.I_i \cap I \neq \emptyset$, output $\langle N.v_i \oplus v, N.I_i \cap I\rangle$.
- If N is not a leaf, then for each i such that $N.I_i \cap I \neq \emptyset$, call $range(N.c_i, I, N.v_i \oplus v)$.

In order to answer a range query over time interval I using an SB-tree rooted at 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]

[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

- 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 $\texttt{NULL} \oplus x = x \oplus \texttt{NULL} = x$ for any x.

For example, when executed on the SB-tree in Fig. 4, $\mathit{range}(N_0, [14, 28), 0)$ 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 Fig. 4, $\mathit{range}(N_0, (-\infty, \infty), 0)$ 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. The running time of *range* is proportional to the number of nodes traversed in the DFT. Suppose that h is the height of the SB-tree and r is the number of leaves that intersect with the given interval. Although the DFT must visit the paths to all r leaves, the number of nodes traversed in the DFT is much less than $h \times r$ because nonleaf nodes are shared among the paths. In fact, the number of nodes traversed is bounded by $O(h + r)$. $O(h)$ bounds the number of nodes on the leftmost and rightmost paths taken by the DFT. $O(r)$ bounds the number of all nodes in between, which form balanced subtrees with $r-2$ leaves that meet the requirement on minimum branching factor. When b and l are large, most nodes are leaves, so the DFT poses very little overhead. In summary, 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.

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 that 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 define the procedure $\mathit{insert}(N, \langle v, I \rangle)$, 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 $N.I_i \cap I \neq \emptyset$:
  - If $N.v_i = v \oplus N.v_i$, do nothing.
  - Otherwise, if $N.I_i \subseteq I$, set $N.v_i$ to $v \oplus N.v_i$.
  - Otherwise, $N.I_i \nsubseteq I$.
    - ◇ If N is not a leaf, call $\mathit{insert}(N.c_i, \langle v, I \rangle)$.
    - ◇ 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 Fig. 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. Suppose we insert tuple $\langle$"Hal", $1, [24, 30) \rangle$ into *Prescription*. Node N_2 in Fig. 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 Sect. 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 $\mathit{insert}(N_0, \langle 1, [17, 47) \rangle)$ on the SB-tree in Fig. 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 $[14, 47)$. Since $N_0.I_2 = [15, 29)$ is not

t will be NULL instead of 0. This change will not affect any of our algorithms.

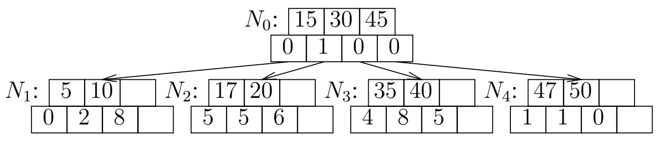

Fig. 5. SB-tree after *insert*

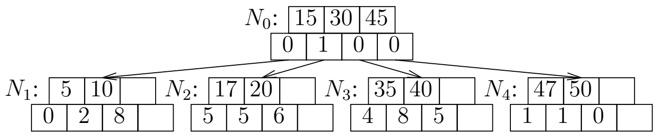

Fig. 6. SB-tree after *delete*

completely covered by $[17, 47)$, we continue with the call $insert(N_2, \langle 1, [17, 47) \rangle)$. Since $N_0.I_3 = [30, 45)$ is completely covered by $[17, 47)$, we simply increment $N_0.v_3$ by 1. Since $N_0.I_4 = [45, \infty)$ is not completely covered by $[17, 47)$, 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 Fig. 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 Sect. 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 Sect. 3.3, the running time of this procedure is $O(h)$, where h is the height of the SB-tree.

For example, consider deleting tuple $\langle$"Ida"$, 1, [17, 47) \rangle$, which we just inserted into *Prescription* in Sect. 3.3. Following the procedure $insert(N_0, \langle -1, [17, 47) \rangle)$ on the SB-tree in Fig. 5, we obtain the SB-tree in Fig. 6. Notice that Fig. 6 is not identical to Fig. 4, the SB-tree before the insertion and the deletion. In particular, the first and the second intervals of N_2 in Fig. 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 Sect. 3.6, we will show how to merge such intervals as a way of compacting the SB-tree.

3.5 Node splitting

As we saw in Sect. 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 complete node splitting procedure *split* is specified formally in Appendix A.

For example, consider executing $insert(N_0, \langle 1, [7, 12) \rangle)$ on the SB-tree in Fig. 4. The resulting SB-tree before any node splitting is shown in Fig. 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 Fig. 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 the SB-tree shown in Fig. 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 split 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 $O(h)$.

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 Sect. 3.4, we saw that a deletion may result in two adjacent leaf intervals with equal aggregate values (Fig. 6). In fact, an insertion could produce the same effect. For instance, in the example of Sect. 3.4, we could have inserted $\langle$"Jay"$, -1, [17, 47) \rangle$ into *Prescription* instead of deleting $\langle$"Ida"$, 1, [17, 47) \rangle$ and obtained exactly the same SB-tree as in Fig. 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 their siblings or merge them with their siblings.

The interval merging procedure *imerge* is used to merge two adjacent leaf intervals with equal aggregate values. The complete *imerge* procedure is specified in Appendix B. The two adjacent intervals may belong to the same leaf, in which case they have equal aggregate values if and only if they have

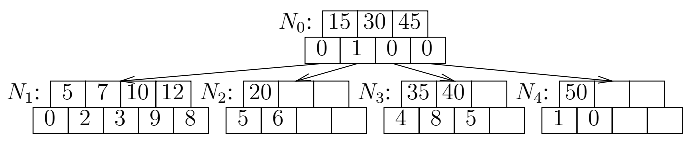

Fig. 7. SB-tree before $split(N_1)$

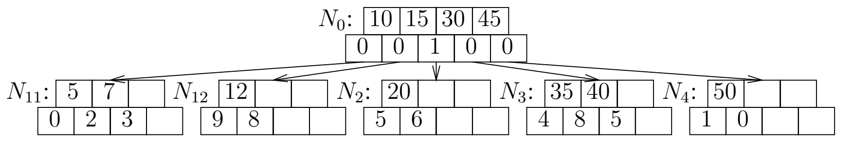

Fig. 8. SB-tree before $split(N_0)$

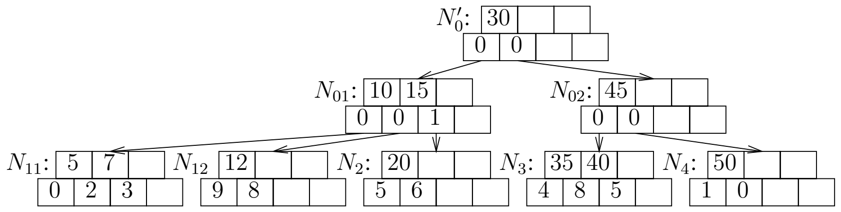

Fig. 9. SB-tree after $split$ completes

equal partial aggregate values in the leaf. The second case is rare and 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. In practice, however, we may choose to ignore the second case since it significantly complicates the procedure and limits concurrent accesses to the SB-tree. In the worst case, the relaxed policy only leads to a slightly bigger SB-tree containing one extra interval per leaf.

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 nonroot 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 has only 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. The procedure $nmerge$ is specified in full in Appendix C.

As a simple example, consider the SB-tree in Fig. 6. We run $imerge$ twice, first on the first and second intervals of N_2 and then on the first and second intervals of N_4. The final result is identical to the SB-tree in Fig. 4. In this example, $nmerge$ is not needed.

As a more complicated example, let us continue with the example in Sect. 3.5. First, we delete the newly inserted tuple by running $insert(N_0, \langle -1, [7, 12) \rangle)$ on the SB-tree in Fig. 9; the result is shown in Fig. 10. We call $imerge$ for the second and third intervals of N_{11} and for the first and 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 third intervals of the parent node N_{01}. The final result is shown in Fig. 12. Notice that the SB-tree in Fig. 12 is not identical to the one we started with in Fig. 4. Nevertheless, they encode exactly the same aggregate.

As an exhaustive example, Fig. 19 illustrates the sequence of snapshots of the SB-tree for aggregate $SumDosage$ as tuples are inserted into $Prescription$ in the order listed in Table 1 and then deleted in the reverse order. The first and last snapshots in Fig. 19 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 Sect. 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

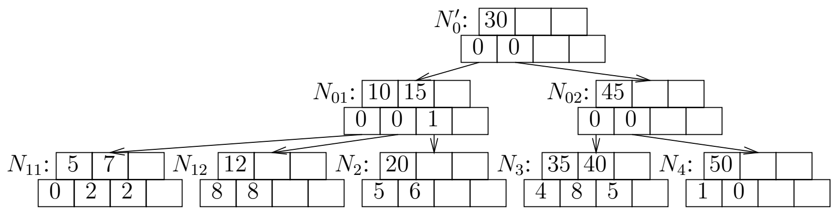

Fig. 10. SB-tree before *imerge*

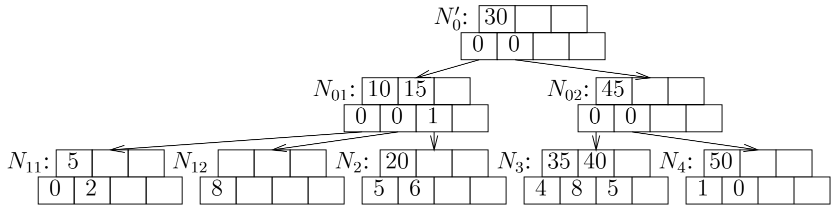

Fig. 11. SB-tree before $nmerge(N_{12})$

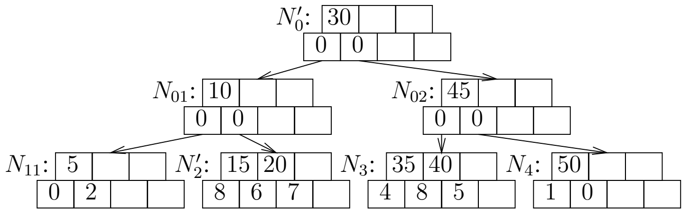

Fig. 12. SB-tree after *imerge* completes

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. On the other hand, as discussed earlier, we may choose to ignore the worst case in a practical implementation for a simpler procedure and better concurrency. To improve concurrency further, we can relax the requirement on minimum branching factor to reduce the need for node merging; this approach is analogous to the *lazy deletion policy* used for B-trees in many commercial database system implementations [7].

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. The output is written out in the format of SB-tree leaves and passed to the bulk loading procedure (Sect. 3.7.1) to construct a new 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 showed in Sect. 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 SB-tree update procedure takes $O(h)$. Furthermore, since the SB-tree is kept compact at all times, $h = \Theta(\log m)$, 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, $h = O(\log n)$, 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 $O(n + m)$:

$O(n)$ bounds the time to traverse the old, noncompact SB-tree; $O(m)$ bounds the time to bulk load the new, compact SB-tree (see Sect. 3.7.1 for details on bulk loading).

3.7 Bulk operations

In this subsection, we consider bulk operations on SB-trees, including bulk loading (constructing an SB-tree from scratch) and bulk updates (processing a set of updates to an existing SB-tree). Because of the similarity between SB-trees and B-trees, many techniques originally developed for B-trees are readily applicable to SB-trees.

3.7.1 Bulk loading

A simple approach is to start with an empty SB-tree and then insert each input tuple in turn, using the standard *insert* procedure. However, this approach is likely to be expensive because every insertion follows one or two paths down the SB-tree. If input tuples arrive in random timestamp order, each insertion will likely follow different paths from the previous ones, resulting in excessive random disk I/Os.

To avoid this problem, we use an efficient batch algorithm that computes the result of a temporal aggregate sorted in ascending timestamp order (such as the endpoint sort algorithm from Sect. 5 for SUM, COUNT, and AVG, or the merge sort algorithm from [15] for MIN and MAX). It is straightforward to modify the batch algorithm to write out SB-tree leaves directly instead of result tuples. As soon as a leaf is produced, we insert the entire leaf into the SB-tree. Since the batch algorithm produces the result in ascending timestamp order, the insertion of a new leaf always follows the rightmost path in the SB-tree. If we can keep the rightmost path of the SB-tree in memory at all times (which only requires h memory blocks), the bulk loading procedure does not need to access disk at all except for writing out the nonleaf nodes. Nonleaf nodes are written to disk whenever they are split out from the rightmost path.

The running time of the bulk loading procedure is dominated by that of the batch algorithm, which is $O(n \log n)$, where n is the size of the input table. Asymptotically, this running time is the same as that of the simple approach where each input tuple is inserted in turn using the standard *insert* procedure. However, counting the number of disk I/Os reveals their difference. The number of I/Os required by an efficient batch algorithm based on external merge sort is typically a small multiple of the total number of input data blocks, while the number of I/Os required by the simple approach is h to $2h$ times the total number of input tuples. Furthermore, the bulk loading procedure can be optimized to perform mostly sequential I/Os, while the simple approach may incur mostly random I/Os.

3.7.2 Bulk updates

Designed with efficient range updates in mind, an SB-tree already provides better update performance than a regular B-tree index on the aggregate result. However, when there are high volumes of data updates, updating an SB-tree immediately for each update can adversely affect the performance of concurrent queries and may be less efficient than updating the index "in bulk." Data warehousing systems usually batch updates and apply them periodically, and this approach may be used on SB-trees as well. Note that for SUM, COUNT, and AVG, deletions can be treated as insertions (Sect. 3.4), and we do not consider deletions for MIN and MAX, so without loss of generality we consider a batch of updates as a batch of insertions.

We can preaggregate a batch of insertions using an efficient batch algorithm (such as those used by bulk loading discussed in Sect. 3.7.1) that produces the result in ascending timestamp order. Each result tuple from the batch algorithm is then inserted into the SB-tree. Since they are sorted by timestamp, multiple successive insertions may follow the same path down the tree, thereby reducing the number of I/Os. In fact, two successive insertions are guaranteed to share at least one path because their intervals share one endpoint (the end of the current interval is also the beginning of the next interval).

The bulk update procedure may appear expensive because the preaggregated result covers the entire time line. However, the number of intervals that make up the time line in this case is still linear in the size of the updates, and an SB-tree has no problem updating long intervals. Furthermore, it is likely that some intervals do not contain any updates; such intervals will have an aggregate value of v_0 (as defined in Sect. 3.2) in the preaggregated result, and they do not need to be inserted into the SB-tree.

A drawback of batching updates is that the aggregate result provided by the SB-tree may be outdated. To overcome this problem, we can borrow techniques developed for B-trees such as the *stepped-merge algorithm* [11]. Instead of applying insertions directly to the main SB-tree, we collect insertions into a current level-0 *run* in memory. When memory is full, we construct an SB-tree for the current level-0 run and write it out to disk, using a procedure similar to bulk loading. Then, a new level-0 run is started. When there are enough level-i runs on disk, we perform a DFT on each of them and merge their outputs together into a larger, level-$(i+1)$ run. The level-i runs are deleted in the process. If the result level-$(i+1)$ run is large enough, we insert the tuples in this run into the main SB-tree in order. Otherwise, we construct an SB-tree for this run and write it out to disk, again using a procedure similar to bulk loading. Queries need to consult any SB-trees for outstanding runs in addition to the main SB-tree, and the results obtained from these SB-trees need to be combined together using the operator $\oplus$ defined in Sect. 3.1. Details of the stepped-merge algorithm can be found in [11]. One notable difference is the handling of deletions. While the B-tree version of the stepped-merge algorithm does not handle deletions, the SB-tree version can handle deletions by treating them as insertions for SUM, COUNT, and AVG.

3.8 Detailed performance analysis

We now present a more detailed analytical performance study of SB-trees, using the number of disk I/Os as the primary cost metric. Suppose that n is the number of tuples in a base table R, n' is the number of distinct interval endpoints in R, and m

Table 6. Number of leaf intervals in an SB-tree

h	Minimum number of leaf intervals	Maximum number of leaf intervals
1	1	683
2	684	349,696
3	175,104	179,044,352
4	44,826,624	$91,670,708,224 > 2^{32}$
5	$11,475,615,744 > 2^{32}$	$> 2^{32}$

is the number of intervals (or output tuples) in the result of the aggregate over R. Each tuple in R has a valid interval with two endpoints, although the endpoints are not necessarily distinct. Therefore, $1 \leq n' \leq 2n$. At the same time, n' is also limited by the granularity of the timestamp. For example, for a base table that records 20 years of history with the granularity of 1 day, n' is less than 20×366. Each distinct interval endpoint corresponds to a potential change in the running aggregate result. Therefore, $1 \leq m \leq n' + 1$. A compact SB-tree has m leaf intervals, while a noncompact SB-tree can have up to $n' + 1$ leaf intervals.

Recall that b denotes the maximum branching factor of a nonleaf node and l denotes the maximum capacity of a leaf node. To simplify subsequent analysis, we assume an average fan-out of f, which is at least $\sqrt{l}$ (for a two-level tree with two leaves), and typically close to $67\% \times b$. The height of an SB-tree, h, is between $\lceil \log_f m \rceil$ (fully compacted) and $\lceil \log_f (n'+1) \rceil$ (not compacted at all). The number of blocks taken by an SB-tree is $\sum_{i=0}^{h-1} f^i = \frac{f^h - 1}{f-1}$. Since f is typically large (in the hundreds), $\frac{f^h-1}{f-1} \approx f^{h-1}$, i.e., most nodes (more than 99%) are leaves. Therefore, the space overhead of storing a compact SB-tree instead of the aggregate result itself mainly comes from leaves that are not full. Since leaves are at least half full, a compact SB-tree typically takes no more than twice the space taken by the aggregate result itself. On the other hand, a noncompact SB-tree may carry significant overhead by storing $n' + 1$ leaf intervals instead of the m intervals in the aggregate result.

Table 6 lists the minimum and maximum numbers of leaf intervals that can be accommodated by a typical SB-tree of height h. In this table, we assume that a block stores 8 KB, a node pointer takes 4 bytes, an interval endpoint takes 4 bytes, and a partial aggregate value takes 8 bytes. In this case, $b = 512$ and $l = 683$. As the table indicates, an SB-tree of 4 levels should be sufficient for any base table (regardless of its size) with 4-byte timestamps because 4 bytes can represent at most 2^{32} unique interval endpoints.

Now we analyze the performance of various SB-tree operations in terms of the number of disk I/Os. Recall that $h = \lceil \log_f m \rceil$ for a compact SB-tree and $h \leq \lceil \log_f (n'+1) \rceil$ for a noncompact SB-tree. Unless otherwise noted, we assume that there are at least h memory blocks available for performing an SB-tree operation.

A lookup (Sect. 3.1) requires one or two paths down the SB-tree and therefore between h and $2h - 1$ I/Os. Only one memory block is required for lookup. A range query (Sect. 3.2) that visits r leaves requires $\sum_{i=1}^{h} r_i$ I/Os, where r_i denotes the number of level-i nodes visited by the range query, which satisfies the following recurrence relation: $r_h = r$, and $r_i \leq \lfloor \frac{r_{i+1}-2}{f} \rfloor + 2$. Solving the recurrence relation, we get $\sum_{i=1}^{h} (\lfloor \frac{r-2}{f^{i-1}} \rfloor + 2) \leq \lceil \frac{f}{f-1}(r-2) \rceil + 2h$, which is roughly $O(h+r)$. As a special case, full reconstruction of the aggregate result requires reading all nodes in the SB-tree, resulting in $\frac{f^h-1}{f-1}$ I/Os. None of the numbers above includes the cost of writing out query results.

An insertion (Sect. 3.3) requires one or two paths down the SB-tree, and therefore between h and $2h-1$ reads. If we force modified nodes to disk, we need between 1 and $2h - 1$ writes. Splitting (Sect. 3.5) in the worst case will result in an additional $4h+1$ writes (when both leaf splits propagate all the way up to the root). For merging (Sect. 3.6), checking adjacent intervals typically does not incur more I/Os but in the worst case may require additional $2h-2$ reads (if the adjacent intervals happen to be on different leaves and the paths from the root to these leaves are almost completely disjoint from the paths taken by the insertion). Moreover, merging requires $2h - 1$ writes in the worst case. Batch merging for `MIN` and `MAX` (Sect. 3.6.1) requires $\frac{f^{h'}-1}{f-1}$ reads to traverse the old, noncompact SB-tree, and $\frac{f^{h''}-1}{f-1}$ writes to write out the new, compact SB-tree, where $h' \leq \lceil \log_f (n'+1) \rceil$ and $h'' = \lceil \log_f m \rceil$.

Bulk loading (Sect. 3.7.1) requires computing the aggregate using a batch algorithm and then constructing the new SB-tree. Batch algorithms based on sorting cost $O(B \cdot \log_M B)$ I/Os, where M is the number of memory blocks available for sorting and $B \approx 2n/l$ is the number of blocks in R. Constructing the new, compact SB-tree requires $\frac{f^h-1}{f-1}$ writes, where $h = \lceil \log_f m \rceil$. For bulk updates (Sect. 3.7.2), preaggregation costs $O(B' \cdot \log_M B')$ I/Os, where B' is the number of blocks taken by the updates. Suppose that the preaggregated result contains m' intervals. Applying the bulk updates to the SB-tree takes $\min(m' \cdot h', \frac{f^{h'}-1}{f-1})$ reads and roughly the same number of writes, where $h' \leq \lceil \log_f (m+m') \rceil$. Note that $\frac{f^{h'}-1}{f-1}$ is an upper bound if $M \geq h'$. With h' memory blocks, we can ensure that we never perform more than one disk read and one disk write for each SB-tree node because the preaggregated result tuples are inserted in order.

4 Cumulative temporal aggregates

In this section, we show how to use SB-trees to compute and maintain cumulative temporal aggregates. Recall from Sect. 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 Sect. 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 approaches: one for `SUM`, `COUNT`, and `AVG` aggregates, and another for `MIN` and `MAX` aggregates.

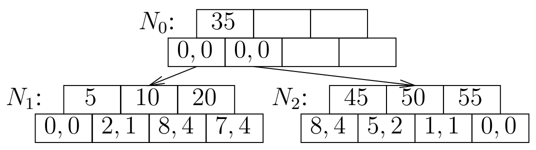

Fig. 13. SB-tree for $AvgDosage_5$

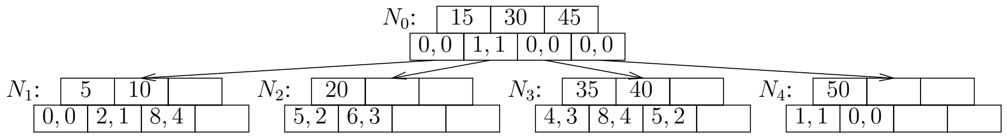

Fig. 14. SB-tree for $AvgDosage$

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 Sect. 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 $\langle v, [I_{start}, I_{end} + w)\rangle$. 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 Sects. 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, Fig. 13 shows the SB-tree index for the cumulative AVG aggregate $AvgDosage_5$ with window offset 5, whose contents are shown in Table 4. Looking up the value of $AvgDosage_5$ at time instant 32 in this SB-tree, we get $N_0.v_1 \oplus N_1.v_4 = \langle 0, 0\rangle \oplus \langle 7, 4\rangle = \langle 7, 4\rangle$. Therefore, the AVG value is $7/4 = 1.75$. For the purpose of comparison, Fig. 14 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 Table 3. 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 offsets such as a day, a week, or 30 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 Fig. 15. The instantaneous SUM aggregates over the two base tables R_1 and R_2 have the same contents. On the other hand,

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	$[10, 40)$

Fig. 15. A cumulative SUM aggregate is not computable from an instantaneous SUM aggregate

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.

4.2.1 Dual SB-trees

One solution is to maintain 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 $lookup(N, t)$. We call this solution *dual SB-trees*. For example, in order to support cumulative AVG aggregates over *Prescription* with arbitrary window offsets, we maintain the SB-tree T shown in Fig. 14 as well as the SB-tree T' shown in Fig. 16. The various operations on dual SB-trees are described next.

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

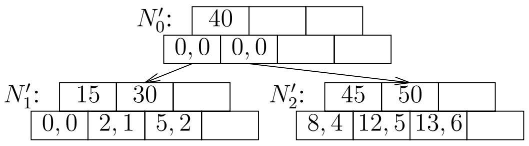

Fig. 16. SB-tree T' for cumulative AVG over *Prescription* with arbitrary window offsets

$$lookup(N,t) \oplus (lookup(N',t) \ominus lookup(N',t-w))$$

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

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

Intuitively, $lookup(N',t) \ominus lookup(N',t-w)$ 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.

For example, let us calculate the value of the cumulative aggregate $AvgDosage_5$ (which has a window offset of 5) at time instant 19 using this method. For the SB-tree T' in Fig. 16, $lookup(N',19) = \langle 0,0\rangle \oplus \langle 2,1\rangle = \langle 2,1\rangle$, and $lookup(N',19-5) = \langle 0,0\rangle \oplus \langle 0,0\rangle = \langle 0,0\rangle$. For the SB-tree T in Fig. 14, $lookup(N,19) = \langle 1,1\rangle \oplus \langle 5,2\rangle = \langle 6,3\rangle$. Therefore, the value of $AvgDosage_5$ at time 19 is $\langle 6,3\rangle \oplus (\langle 2,1\rangle \ominus \langle 0,0\rangle) = \langle 8,4\rangle$, which is consistent with the answer we obtained in Sect. 4.1 from the SB-tree built specifically for $AvgDosage_5$ in Fig. 13.

Range query A range query over interval $I = [I_{start}, I_{end})$ is answered by combining the results of two range queries $range(N', [I_{start} - w, I_{end}), v_0)$ and $range(N, I, v_0)$, where v_0 is the aggregate-dependent value defined in Sect. 3.2. We can avoid producing the intermediate results by coordinating the DFTs on T and T'.

Insertion, deletion, splitting, and merging When there is an insertion into the base table, we maintain T as discussed in Sect. 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 captured by the pair $\langle v, [I_{end}, \infty)\rangle$ since $lookup(N',t)$ 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 Sect. 3. Deletions are handled 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.

4.2.2 Joint SB-trees

An alternative to dual SB-trees is an SB-tree variant called the *Joint SB-tree*, or *JSB-tree* for short. In a JSB-tree, each interval is associated with a pair of partial aggregate values v^- and v^+. Intuitively, v^- is computed over all tuples that are valid strictly before a particular time instant (as in T' in Sect. 4.2.1), and v^+ is computed over all tuples that are valid strictly after the time instant. Each JSB-tree also includes a "global" aggregate value v^{global}, which is computed over all tuples regardless of their valid periods. As an example, the JSB-tree shown in Fig. 17 supports cumulative AVG aggregates over *Prescription* with arbitrary window offsets. Next, we describe the operations on JSB-trees.

Lookup Using a JSB-tree rooted at N^J, we take the following steps to look up the value of a cumulative aggregate with window offset w at time instant t:

- Follow the path from N^J to the leaf interval containing $t-w$ and accumulate all v^- values along this path. We refer to this procedure as $lookup^-(N^J, t-w)$. It is identical to regular SB-tree *lookup* except that we accumulate v^- values rather than v values.
- Follow the path from N^J to the leaf interval containing t and accumulate all v^+ values along the path. We refer to this procedure as $lookup^+(N^J, t)$. It is identical to regular SB-tree *lookup* except that we accumulate v^+ values rather than v values.
- The final answer is calculated by: $$v^{global} \ominus (lookup^-(N^J, t-w) \oplus lookup^+(N^J, t))$$

Intuitively, $lookup^-(N^J, t-w)$ computes the aggregate value over all tuples that are valid strictly before $t-w$, and $lookup^+(N^J, t)$ computes the aggregate value over all tuples that are valid strictly after t. We subtract these two terms from v^{global}, which is computed over all tuples, and the result is the aggregate value computed over all tuples that are valid at some point during $[t-w, t]$.

For example, let us calculate the value of $AvgDosage_5$ ($w = 5$) at time instant 19 using this method. According to the JSB-tree in Fig. 17, $v^{global} = \langle 13,6\rangle$; $lookup^-(N_0^J, 19-5) = \langle 4,1\rangle \oplus \langle 1,1\rangle = \langle 5,2\rangle$; and $lookup^+(N_0^J, 19) = \langle 0,0\rangle \oplus \langle 2,1\rangle = \langle 2,1\rangle$. Therefore, the final answer is $\langle 13,6\rangle \ominus (\langle 5,2\rangle \oplus \langle 2,1\rangle) = \langle 6,3\rangle$, which is correct and consistent with the answer we obtained using dual SB-trees in Sect. 4.2.1.

Both $lookup^-(N^J, t-w)$ and $lookup^+(N^J, t)$ are $O(h)$, where h is the height of the JSB-tree. Note that when window offset w is small, $lookup^-(N^J, t-w)$ and $lookup^+(N^J, t)$ will share a large portion of the path from the root. In this case, we can optimize JSB-tree lookup by combining $lookup^-$ and $lookup^+$ or by caching the JSB-tree nodes accessed by $lookup^-$ in main memory for $lookup^+$. However, these optimizations do not change the asymptotic complexity of JSB-tree lookup.

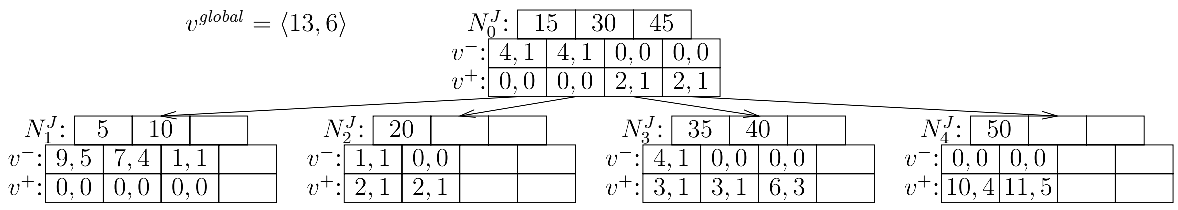

Fig. 17. JSB-tree T^J for cumulative AVG aggregates over *Prescription* with arbitrary window offsets

Table 7. Comparison of dual SB-trees and JSB-trees

	Dual SB-trees	Paths	JSB-tree	Paths
Lookup of cumulative aggregate at t with window offset w	$lookup(N, t) \oplus$ $(lookup(N', t) \ominus$ $lookup(N', t - w))$	$\cdots 1$ $\cdots 1$ $\cdots 1$	$v^{global} \ominus$ $(lookup^-(N^J, t - w) \oplus$ $lookup^+(N^J, t))$	 $\cdots 1$ $\cdots 1$
Lookup of instantaneous aggregate at t	$lookup(N, t)$	$\cdots 1$	$v^{global} \ominus$ $(lookup^-(N^J, t) \oplus$ $lookup^+(N^J, t))$	 $\cdots$ 1 $\cdots$
Insert of $\langle v, I\rangle$ $I = [I_{start}, I_{end})$	$insert(N, \langle v, I\rangle)$ $insert(N', \langle v, (-\infty, I_{start})\rangle)$	$\cdots 2$ $\cdots 1$	$insert^-(N^J, (-\infty, I_{start}))\rangle)$ $insert^+(N^J, [I_{end}, \infty))\rangle)$ $v^{global} = v^{global} \oplus v$	$\cdots 1$ $\cdots 1$

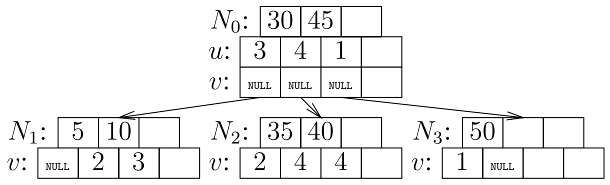

Fig. 18. MSB-tree for cumulative MAX over *Prescription* with arbitrary window offsets

With a JSB-tree, there is no longer any need to maintain an SB-tree for the instantaneous aggregate. After all, an instantaneous aggregate is just a cumulative aggregate with window offset $w = 0$. Using a JSB-tree rooted at N^J, we can calculate the instantaneous aggregate value at time t as:

$$v^{global} \ominus (lookup^-(N^J, t) \oplus lookup^+(N^J, t))$$

Note that $lookup^-(N^J, t)$ and $lookup^+(N^J, t)$ can be computed in one pass down the JSB-tree because they follow the same path from N^J to the leaf interval containing t. Thus, JSB-trees are comparable to SB-trees in efficiency for handling instantaneous aggregates. The end of Sect. 4.2.2 provides a more comprehensive and detailed comparison of JSB-trees and dual SB-trees.

Range query A range query over interval $I = [I_{start}, I_{end})$ is computed from two range queries $range^-(N^J, [I_{start} - w, I_{end} - w), v_0)$ and $range^+(N^J, I, v_0)$, where v_0 is the aggregate-dependent value defined in Sect. 3.2. Procedures $range^-$ and $range^+$ are identical to regular SB-tree $range$ except that they operate on v^- and v^+ Values, respectively, instead of v values. Computing $range^-(N^J, [I_{start} - w, I_{end} - w), v_0)$ and $range^+(N^J, I, v_0)$ together requires one DFT on the JSB-tree within the range $[I_{start} - w, I_{end})$.

Insertion, deletion, splitting, and merging When there is an insertion with effect $\langle v, I\rangle$, where $I = [I_{start}, I_{end})$, we take the following steps to update the JSB-tree rooted at N^J:

- Call $insert^-(N^J, \langle v, (-\infty, I_{start})\rangle)$.
- Call $insert^+(N^J, \langle v, [I_{end}, \infty)\rangle)$.
- Set v^{global} to $v^{global} \oplus v$.

Again, procedures $insert^-$ and $insert^+$ are identical to regular SB-tree $insert$ except that they operate on v^- and v^+ values, respectively, instead of v values.

The insertion cost is dominated by $insert^-$ and $insert^+$, both of which are $O(h)$, where h is the height of the JSB-tree. Furthermore, note that $insert^-$ and $insert^+$ are always invoked with intervals that are unbounded at one end, so each invocation of $insert^-$ or $insert^+$ updates only one path in the JSB-tree. In comparison, each invocation of regular SB-tree $insert$ can update two paths in the SB-tree.

We treat deletions as insertions with negative effects, as for SB-trees. We also use SB-tree splitting and merging procedures for JSB-trees by defining v as the pair $\langle v^-, v^+\rangle$ for the purpose of these procedures. Furthermore, we define $\langle v_1^-, v_1^+\rangle \oplus \langle v_2^-, v_2^+\rangle$ to be $\langle v_1^- \oplus v_2^-, v_1^+ \oplus v_2^+\rangle$. For example, recall that two adjacent leaf intervals can be merged only if the accumulated v values along the two paths turn out to be equal. For a JSB-tree, this condition would mean that the two intervals' accumulated v^- values and v^+ values are respectively equal.

As a complete example, Fig. 20 shows the sequence of snapshots of the JSB-tree for cumulative AVG aggregates over *Prescription* as tuples are inserted in the order listed in Table 1; Fig. 21 shows the sequence of snapshots as tuples are deleted in the reverse over.

Discussion A comparison between dual SB-trees and JSB-trees is shown in Table 7. Although these two approaches have the same asymptotic complexity, JSB-trees appear to be more

Table 8. 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 [20]	All	Disk	$O(n^2)$	No	No	No
Balanced tree [15]	`SUM`/`COUNT`/`AVG`	Memory	$O(n \log n)$	No	No	No
Endpoint sort (Sect. 5)	`SUM`/`COUNT`/`AVG`	Disk	$O(n \log n)$	No	No	No
Merge sort [15]	`MIN`/`MAX`	Disk	$O(n \log n)$	No	No	No
Aggregation tree [12]	All	Memory	$O(n^2)$	$O(n)$	$O(n)$ (no if k-ordered)	No
SB-tree (Sects. 3, 4.1)	All	Disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Fixed window offset
Dual SB-trees and JSB-tree (Sect. 4.2)	`SUM`/`COUNT`/`AVG`	Disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Arbitrary window offset
MSB-tree (Sect. 4.3)	`MIN`/`MAX`	disk	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Arbitrary window offset

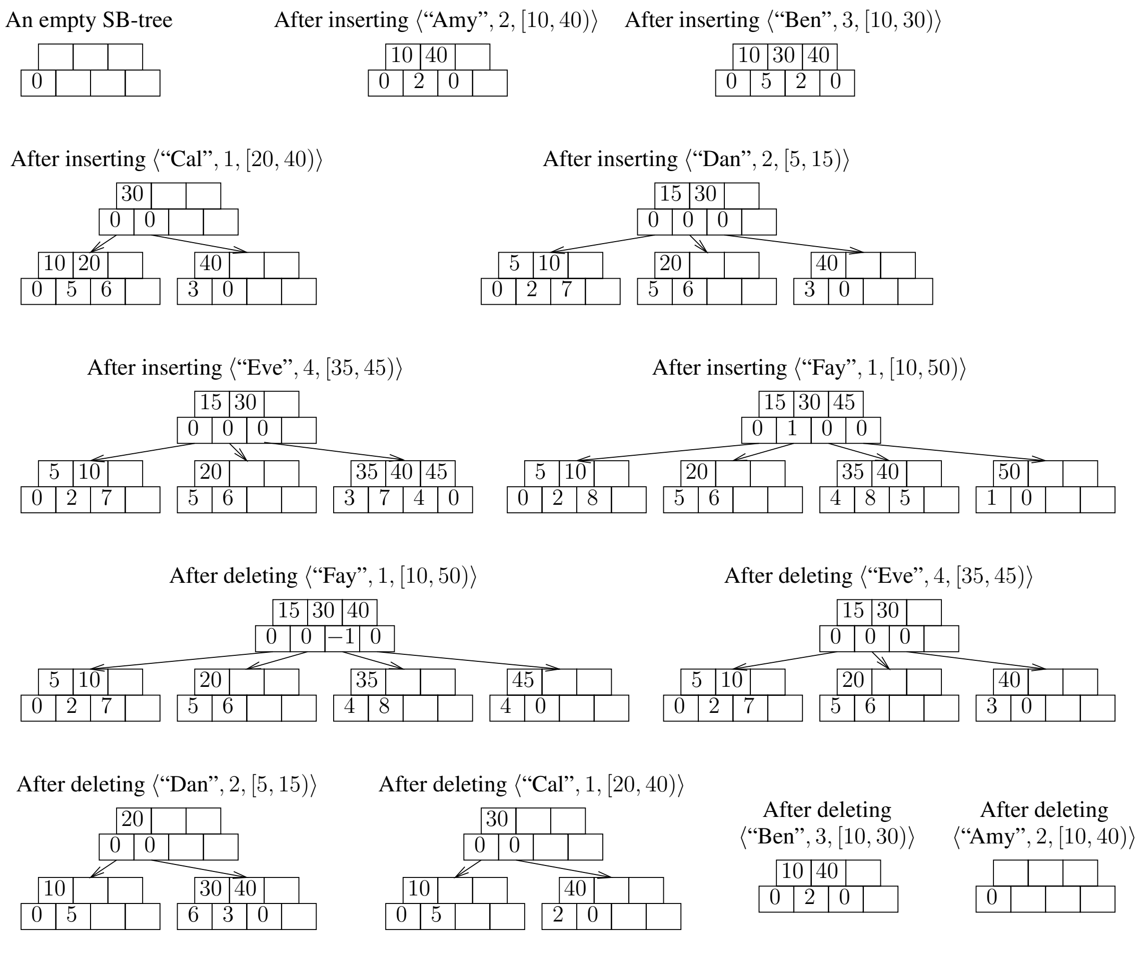
An empty SB-tree
After inserting ⟨"Amy", 2, [10, 40)⟩
After inserting ⟨"Ben", 3, [10, 30)⟩
After inserting ⟨"Cal", 1, [20, 40)⟩
After inserting ⟨"Dan", 2, [5, 15)⟩
After inserting ⟨"Eve", 4, [35, 45)⟩
After inserting ⟨"Fay", 1, [10, 50)⟩
After deleting ⟨"Fay", 1, [10, 50)⟩
After deleting ⟨"Eve", 4, [35, 45)⟩
After deleting ⟨"Dan", 2, [5, 15)⟩
After deleting ⟨"Cal", 1, [20, 40)⟩
After deleting ⟨"Ben", 3, [10, 30)⟩
After deleting ⟨"Amy", 2, [10, 40)⟩

Fig. 19. SB-tree for $SumDosage$

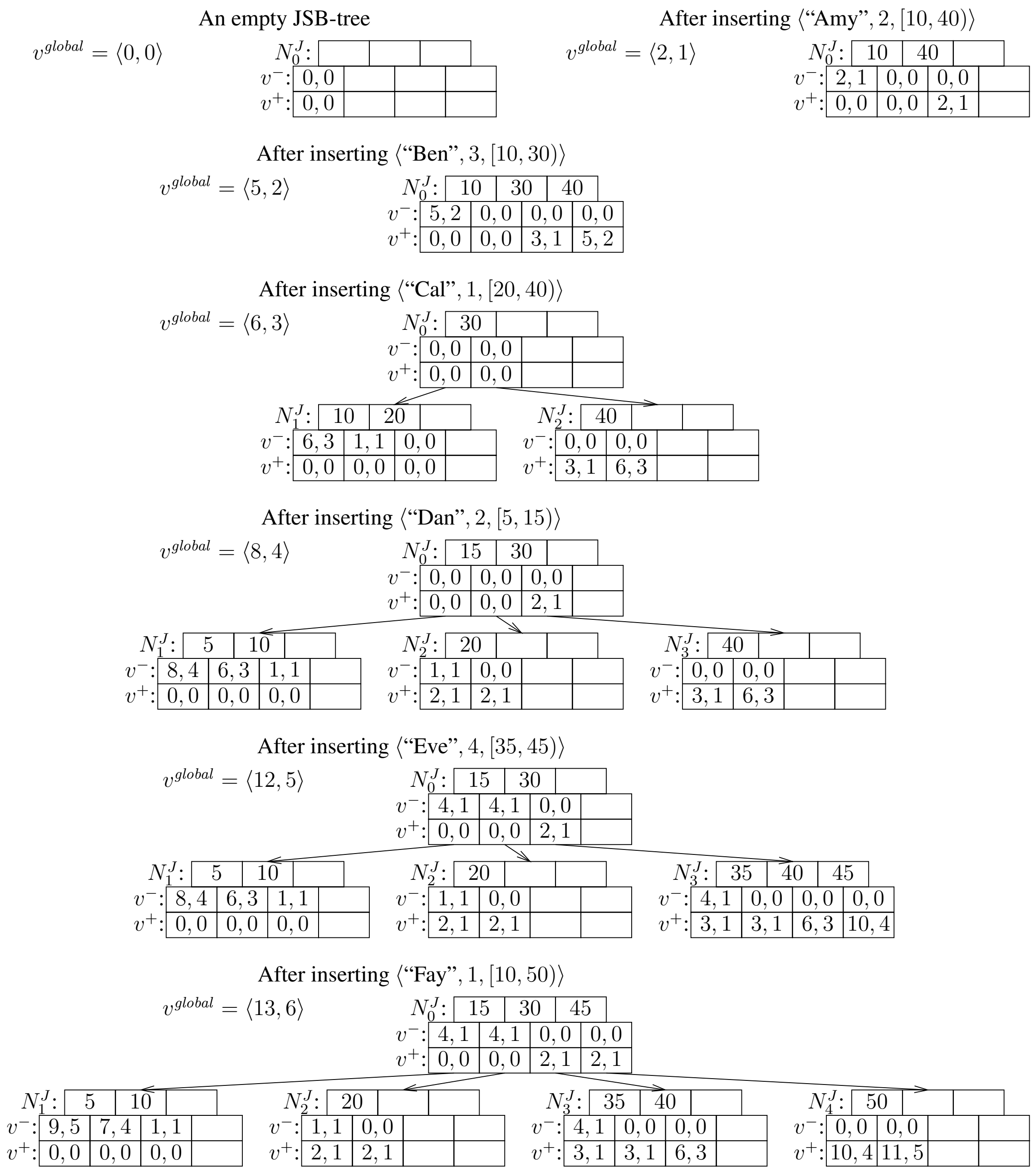

Fig. 20. JSB-tree for cumulative AVG aggregates over *Prescription* with arbitrary window offsets

efficient than dual SB-trees because JSB-tree operations typically traverse fewer paths. However, there is a trade-off. Since each interval in a JSB-tree is associated with two partial aggregate values instead of one, a JSB-tree node can hold fewer intervals than an SB-tree node. Consequently, JSB-trees may be taller than SB-trees, and JSB-tree operations may have to traverse longer paths than SB-tree operations. Truly understanding the trade-offs would require a performance study over realistic data.

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 $range(N, [t-w, t], v_0)$, where $v_0 = \texttt{NULL}$ (see Sect. 3.2); the answer we are looking for is the MIN or MAX value of all the output tuples. Recall from Sect. 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 nonleaf nodes. For each interval $N.I_i$ in a nonleaf node N, we store a "u" value denoted

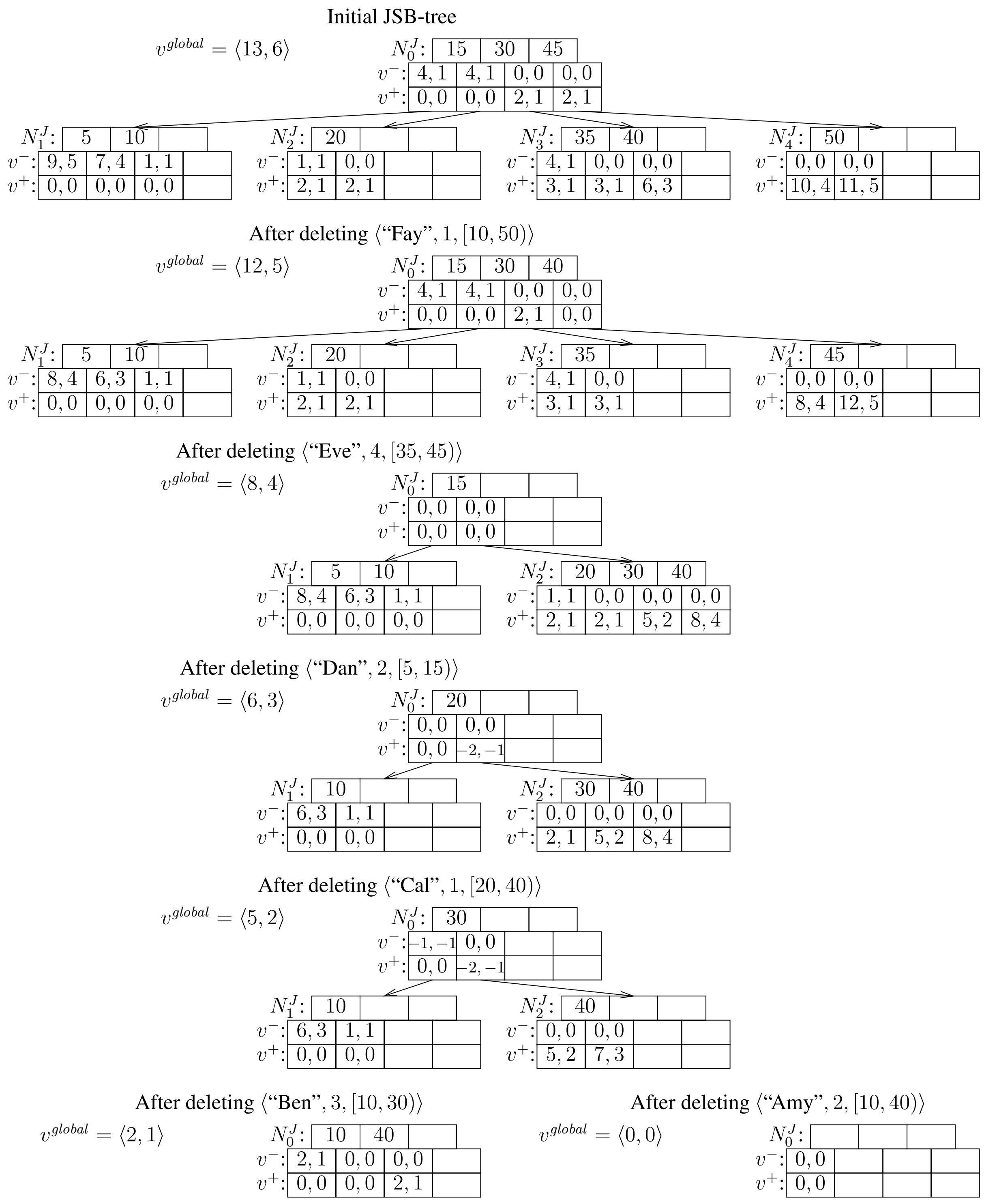

Fig. 21. JSB-tree for cumulative AVG aggregates over *Prescription* with arbitrary window offsets

$N.u_i$ in addition to the "v" value $N.v_i$. Intuitively, $N.u_i$ is the precomputed result of the aggregate over all v values in the subtree rooted at $N.c_i$. There is no need to store any u value for a leaf interval. We call this new index structure an *MSB-tree* (for MIN/MAX SB-tree).

To describe the idea of MSB-trees, we use MAX aggregates as an example; the case of MIN aggregates is analogous. Suppose that N is not a leaf. The MAX value during $N.I_i$ should be the maximum of all v values along the paths from the root to leaf intervals contained in $N.I_i$. All such paths originate from the root and go through $N.c_i$. From $N.c_i$, the paths diverge and span the entire subtree rooted at $N.c_i$. We can easily compute the maximum of all v values along the shared section of these paths. The maximum of all v values in the subtree would be expensive to compute on the fly, so we incrementally compute and maintain this maximum as $N.u_i$.

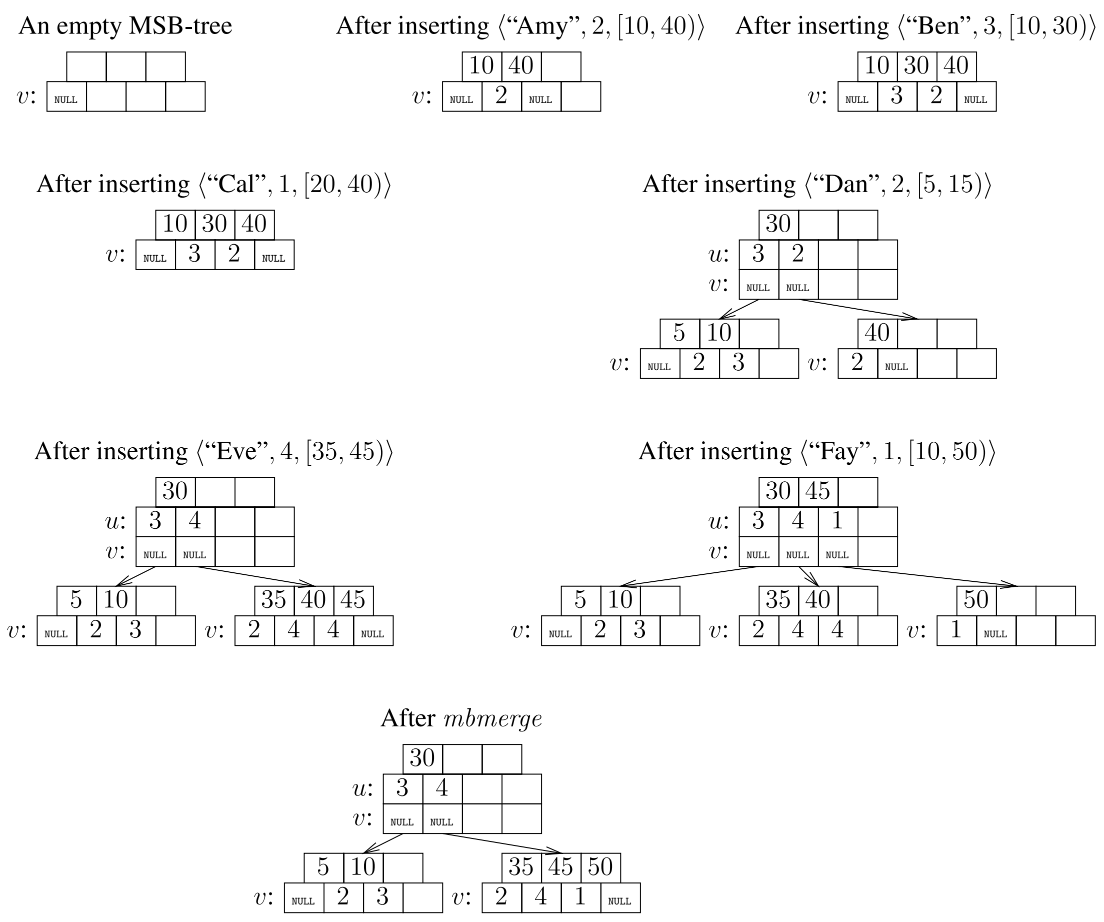
An empty MSB-tree
After inserting ⟨"Amy", 2, [10, 40)⟩
After inserting ⟨"Ben", 3, [10, 30)⟩
After inserting ⟨"Cal", 1, [20, 40)⟩
After inserting ⟨"Dan", 2, [5, 15)⟩
After inserting ⟨"Eve", 4, [35, 45)⟩
After inserting ⟨"Fay", 1, [10, 50)⟩
After *mbmerge*
u:
v:
NULL

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

For example, Fig. 18 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 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 $u = v_0 =$ `NULL` (Sect. 3.2). The definition of $mlookup(N, t, w, u)$ follows.

- Let $u_{my} = u$.
- For each i such that $N.I_i \cap [t - w, t] \neq \emptyset$:
  - If N is a leaf, set $u_{my} = u_{my} \oplus N.v_i$.
  - Otherwise, N is not a leaf:
    - If $u_{my} = u_{my} \oplus N.u_i \oplus N.v_i$, do nothing.
    - Otherwise, if $N.I_i \subseteq [t - w, t]$, set $u_{my} = u_{my} \oplus N.u_i \oplus N.v_i$.
    - Otherwise, set $u_{my} = mlookup(N.c_i, t, w, u_{my} \oplus N.v_i)$.
- 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, Fig. 18 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 Table 5.

Notice that this procedure is almost identical in structure to the SB-tree *insert* procedure defined in Sect. 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 interval $I = [I_{start}, I_{end})$ can be answered easily by a single pass over the result of $range(N, [I_{start} - w, I_{end}), v_0)$, where N is the root of the MSB-tree and $v_0 =$ `NULL` (Sect. 3.2). However, *range* does not take advantage of the u values stored in nonleaf nodes. We can optimize the procedure by making use of the u values to avoid producing any intermediate result.

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 $N.I_i \cap I \neq \emptyset$:
  - If N is not a leaf, set $N.u_i = v \oplus N.u_i$.
  - If $N.v_i = v \oplus N.v_i$, do nothing.
  - Otherwise, if $N.I_i \subseteq I$, set $N.v_i$ to $v \oplus N.v_i$.
  - 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 $\langle v, I\rangle$.

This procedure is identical to the SB-tree *insert* procedure defined in Sect. 3.3 except for the additional line at the beginning of the loop. This line catches all inserts through $N.c_i$ and guarantees that $N.u_i$ always equals the MIN or MAX v value in the entire subtree rooted at $N.c_i$. We update the u value for any nonleaf 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 after *minsert*, we split it by calling $msplit(N)$. The *msplit* procedure is identical to the SB-tree *split* procedure defined in Sect. 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 Sect. 3.6. The analysis of running time is similar to the SB-tree case covered at the end of Sect. 3.6. Also, as discussed in Sect. 3.4, we do not handle deletions for MIN and MAX aggregates.

As a complete example, Fig. 22 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 Table 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 nonleaf 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 Sect. 6, we will provide a comparison chart of all the temporal aggregation algorithms discussed in Sect. 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 or disk-based. For completeness, we quickly present a disk-based version of the balanced-tree algorithm in [15] for instantaneous aggregates SUM, COUNT, and AVG. The [15] 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 this tuple's valid interval (Sect. 3.3). Generate two tuples $\langle v, I_{start}\rangle$ and $\langle v_0 \ominus v, I_{end}\rangle$, where v_0 and $\ominus$ are as defined in Sects. 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 $\langle v_1 \oplus v_2, t\rangle$ (or remove them if $v_1 \oplus v_2 = v_0$), where $\oplus$ is as defined in Sect. 3.1.
- 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 $v_{my} = v_{my} \oplus v$.

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 points and endpoints of all valid intervals together. As an example, for base table *Prescription* in Table 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 the fact that the prescriptions for Amy, Ben, and Fay all start at time 10. The third tuple, $\langle -2, 15\rangle$, reflects the fact 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 points or endpoints.

It is straightforward to modify the sorting algorithm so that the generation of $\langle v, t\rangle$ tuples and the computation of v_{my} do not require additional passes. For example, consider the classic multipass external merge sort. The generation of $\langle v, t\rangle$ tuples can be done while reading R for the first time.

The computation of v_{my} can be carried out for each sorted run as it is being generated.

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 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 structure for supporting temporal aggregates in data warehouses.

We have also described four approaches to handling cumulative temporal aggregates. The first approach requires only a slight change to 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 and third approaches both handle cumulative `SUM`, `COUNT`, and `AVG` aggregates: the second approach uses a pair of SB-trees, and the third approach uses an extension of the SB-tree called the JSB-tree. The fourth approach uses another extension of the SB-tree called the MSB-tree to handle cumulative `MIN` and `MAX` aggregates. Compared to the basic SB-tree, the last three 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 Table 8, we compare our SB-tree algorithms and the endpoint sort algorithm of Sect. 5 with the other temporal aggregation algorithms discussed in Sect. 2. For simplicity of presentation, Table 8 provides only rough upper bounds on the running times of algorithms; please refer to the appropriate sections of this paper for detailed analyses.

SB-trees, like most other algorithms compared in Table 8, do not handle the `DISTINCT` versions of `SUM`, `COUNT`, and `AVG`. The reason is that partial or running aggregate values alone are insufficient for determining whether a particular input value should contribute to the aggregate because it has not been seen before. It would be interesting to study the incremental computation and maintenance of `DISTINCT` aggregates, which require a significant amount of state to be maintained. As future work, we also plan to implement the SB-tree and its variants in a disk-based environment and measure their performance with real-world applications. Finally, we plan to consider concurrency control algorithms for the SB-tree and its variants.

Acknowledgements. We thank Jeff Ullman for proposing the idea of JSB-trees as an alternative to dual SB-trees. We also thank Hector Garcia-Molina, Carlo Zaniolo, and anonymous reviewers for their valuable comments on SB-trees. Finally, we are grateful to Marios Hadjieleftheriou for his recent implementation of a Java SB-Tree library [9].

References

1. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Informat 1:173–189
2. Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P (1996) An asymptotically optimal multiversion B-tree. VLDB J 5(4):264–275
3. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. MIT Press, Cambridge, MA
4. Epstein R (1979) Techniques for processing of aggregates in relational database systems. Technical Report UCB/ERL M7918, University of California, Berkeley, CA
5. Geffner S, Agrawal D, El Abbadi A (2000) The dynamic data cube. In: Proceedings of the 2000 international conference on extending database technology, Konstanz, Germany, March 2000, pp 237–253
6. Gendrano JAG, Huang BC, Rodrigue JM, Moon B, Snodgrass RT (1999) Parallel algorithms for computing temporal aggregates. In: Proceedings of the 1999 international conference on data engineering, Sydney, Australia, March 1999, pp 418–427
7. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Morgan Kaufmann, San Mateo, CA
8. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, Boston, June 1984, pp 47–57
9. Hadjieleftheriou M (2001) Java SB-tree library. `http://www.cs.ucr.edu/~marioh/sbtree/`.
10. Ho CT, Agrawal R, Megiddo N, Srikant R (1997) Range queries in OLAP data cubes. In: Proceedings of the 1997 ACM SIGMOD international conference on management of data, Tucson, AZ, June 1997, pp 73–88
11. Jagadish HV, Narayan PPS, Seshadri S, Sudarshan S, Kanneganti R (1997) Incremental organization for data recording and warehousing. In: Proceedings of the 1997 international conference on very large data bases, Athens, Greece, August 1997, pp 16–25
12. Kline N, Snodgrass RT (1995) Computing temporal aggregates. In: Proceedings of the 1995 international conference on data engineering, Taipei, Taiwan, March 1995, pp 222–231
13. Kolovson CP, Stonebraker M (1991) Segment indexes: dynamic indexing techniques for multi-dimensional interval data. In: Proceedings of the 1991 ACM SIGMOD international conference on management of data, Denver, CO, May 1991, pp 138–147
14. Kriegel HP, Pötke M, Seidl T (2000) Managing intervals efficiently in object-relational databases. In: Proceedings of the 2000 international conference on very large data bases, Cairo, Egypt, September 2000, pp 407–418
15. Moon B, Lopez IFV, Immanuel V (2000) Scalable algorithms for large temporal aggregation. In: Proceedings of the 2000 international conference on data engineering, San Diego, March 2000, pp 145–154
16. Preparata FP, Shamos MI (1985) Computational geometry: an introduction. Springer, Berlin Heidelberg New York
17. Salzberg B, Tsotras VJ (1999) Comparison of access methods for time-evolving data. ACM Comput Surv 21(2):158–221
18. Snodgrass RT (ed) (1995) The TSQL2 temporal query language. Kluwer, Boston
19. Snodgrass RT, Gomez S, McKenzie LE (1993) Aggregates in the temporal query language TQuel. IEEE Trans Knowl Data Eng 5(5):826–842

20. Tuma PA (1992) Implementing historical aggregates in TempIS. Master's thesis, Wayne State University, Detroit, MI
21. Yang J, Widom J (1998) Maintaining temporal views over non-temporal information sources for data warehousing. In: Proceedings of the 1998 international conference on extending database technology, Valencia, Spain, March 1998, pp 389–403
22. Yang J, Widom J (2000) Temporal view self-maintenance in a warehousing environment. In: Proceedings of the 2000 international conference on extending database technology, Konstanz, Germany, March 2000, pp 395–412
23. Yang J, Widom J (2001) Incremental computation and maintenance of temporal aggregates. In: Proceedings of the 2001 international conference on data engineering, Heidelberg, Germany, April 2001
24. Ye X, Keane JA (1997) Processing temporal aggregates in parallel. In: Proceedings of the 1997 IEEE international conference on systems, man, and cybernetics, Orlando, October 1997, pp 1373–1378
25. Zhang D, Markowetz A, Tsotras V, Gunopulos D, Seeger B (2001) Efficient computation of temporal aggregates with range predicates. In: Proceedings of the 2001 ACM symposium on principles of database systems, Santa Barbara, May 2001

Appendix A Node splitting procedure

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 $\mathit{split}(N)$, which reorganizes the SB-tree to deal with the overflow at node N.

- Split N into N_1 and N_2, such that:
  - ○ N_1 contains the first $\lceil \frac{n}{2} \rceil$ intervals of N; that is, N_1 contains time instants $N.t_1, \ldots, N.t_{\lceil \frac{n}{2} \rceil - 1}$, partial 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 $N.t_{\lceil \frac{n}{2} \rceil + 1}, \ldots, N.t_{n-1}$, partial 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 Sect. 3.2. (Recall from the beginning of Sect. 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, move each interval 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 $\mathit{split}(N')$.

Appendix B Interval merging procedure

As discussed in Sect. 3.6, the interval merging procedure imerge has two cases:

- The two adjacent intervals belong to the same leaf N. Suppose 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 $\mathit{nmerge}(N)$.
- 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 $\mathit{lookup}(N.c_k, \mathit{start}(N_1.I_j)) = \mathit{lookup}(N.c_{k+1}, \mathit{start}(N_2.I_1))$ (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 $\mathit{nmerge}(N_2)$.

Appendix C Node merging procedure

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

- If N is the root:
  - ○ If N has exactly one child, make $N.c_1$ the new root, set $N.c_1.v_i = N.v_1 \oplus N.c_1.v_i$ 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 that 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 $N.v_i = N_p.v_k \oplus N.v_i$.

- ◇ In N_p, set $N_p.v_k = v_0$, where v_0 is the value defined in Sect. 3.2.
- ◇ In N, set $N.t_{\lceil\frac{n}{2}\rceil-1} = N_p.t_k$ and $N.v_{\lceil\frac{n}{2}\rceil} = N_p.v_{k+1} \oplus N'.v_1$. Furthermore, if N is not a leaf, set $N.c_{\lceil\frac{n}{2}\rceil} = N'.c_1$.
- ◇ 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 that 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 $N.v_i = N_p.v_{k+1} \oplus N.v_i$.
- ◇ In N_p, set $N_p.v_{k+1} = v_0$, where v_0 is the value defined in Sect. 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 $N.t_1 = N_p.t_k$, $N.v_1 = N_p.v_k \oplus N'.v_j$, 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 $N'.v_i = N_p.v_k \oplus N_1.v_i$.
    For $j_1 < i \leq j_1+j_2$, set $N'.v_i = N_p.v_{k+1} \oplus N_2.v_{i-j_1}$.
  - – 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 Sect. 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 $nmerge(N_p)$.