Temporal and Spatio-Temporal Aggregations over Data Streams using Multiple Time Granularities

Donghui Zhang [a] *, Dimitrios Gunopulos [a] †, Vassilis J. Tsotras [a] ‡, Bernhard Seeger [b]

[a]Computer Science Department, University of California, Riverside, CA 92521.
{donghui,dg,tsotras}@cs.ucr.edu

[b] Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany.
seeger@Mathematik.Uni-Marburg.de

Temporal and spatio-temporal aggregations are important but costly operations for applications that maintain time-evolving data (data warehouses, temporal databases, etc.). In this paper we examine the problem of computing such aggregates over data streams. The aggregates are maintained using multiple levels of temporal granularities: older data is aggregated using coarser granularities while more recent data is aggregated with finer detail. We present specialized indexing schemes for dynamically and progressively maintaining temporal and spatio-temporal aggregates. Moreover, these schemes can be parameterized. The levels of granularity as well as their corresponding index sizes (or validity lengths) can be dynamically adjusted. This provides a useful trade-off between aggregation detail and storage space. Analytical and experimental results show the efficiency of the proposed structures. We first address the temporal aggregation problem. A general framework of aggregating at multiple time granularities is then proposed. Finally we show how to utilize this framework to solve the range temporal and spatio-temporal aggregation problems.

Keywords: temporal aggregation; range-temporal aggregation; spatio-temporal aggregation; data stream; multiple time granularities

1. Introduction

With the rapid increase of historical data in data warehouses, temporal aggregation has become predominant operators for data analysis. Computing temporal aggregates is a significantly more intricate problem than traditional aggregation. Each database tuple has an attribute value (e.g. the dosage of a prescription) and is accompanied by a time interval during which the attribute value is valid. Consequently, the value of a tuple attribute affects the aggregate computation for all those instants included in the tuple's time interval. An *instantaneous* temporal aggregate is the aggregate value of all tuples whose intervals contain a given time instant. A *cumulative* temporal aggregate is the aggregate value of all tuples whose intervals intersect a given time interval. For example, "find the total number of phone calls made in 1999". In the rest we concentrate on cumulative aggregates since they are more general; thus the term "temporal aggregation" implies "cumulative temporal aggregation". Furthermore, in this paper we focus on the SUM aggregate but our solutions apply to COUNT and AVG as well.

Many approaches have been recently proposed to address temporal aggregation queries [28,19, 29,11,22,30,31]. They are classified into two categories: approaches that compute a temporal aggregate when the aggregate is requested (usually by sweeping through related data) and those that maintain a specialized aggregate index [30,31]. The latter approaches dynamically precompute aggregates and store them appropriately in the specialized index. This leads to less space (since the aggregation index is typically much smaller

*Corresponding author. Tel.: +1-909-787-2838.

†This work was partially supported by NSF CAREER Award 9984729, NSF IIS-9907477, the DoD and AT&T.

‡This work was partially supported by NSF grants IIS-9907477, EIA-9983445, and the Department of Defense.

than the actual data) as well as much faster query times (an ad-hoc aggregate is computed by simply traversing a path in the index). In particular, [30] proposed the SB-tree, an elegant index used to solve the *scalar* temporal aggregation problem: the aggregation involves all tuples whose intervals intersect the query time interval. [31] introduced the MVSB-tree and solved a more general problem, the *range* temporal aggregation, where the aggregate is taken over all tuples intersecting the query time interval and having keys in a query specified key range.

However, all previous works assume that the temporal aggregation is expressed in a single time granularity. Usually this granularity is the same as the granularity used to store the time attributes. Recently, there has been much research on multiple time granularities [4,9,8,6,3]. In many data warehousing query languages, e.g. Microsoft's MDX, one query can return results at multiple granularities. Consider a database tracking the phone calls records and let the time granularity be in seconds. Each phone call record is accompanied by an interval [*start*, *end*) (both in seconds) indicating the time period this call took place. A temporal aggregate example is: "find the total number of phone calls made between 12:30:01 and 15:30:59 today". While aggregating per second may be necessary for queries on the recent enterprise history (say within 10 days), for many applications it is not crucial when querying the remote history (for example, data older than a year ago). In the latter case, the aggregation at a coarser time granularity (e.g., per minute or per day) may be satisfactory enough. The ability to aggregate using coarser granularities for older data is crucial for applications that accumulate large amounts of data. As an example, [18] cites that a major telecommunications company collects 75GB of detailed call data every day or 27TB a year. With this huge amount of source data, even the specialized aggregation index under a single time granularity will soon grow too large.

Saving storage space is especially useful for applications under the stream model, which was formalized recently by [17]. A stream is an ordered sequence of points that are read in increasing order. The performance of an algorithm that operates on streams is measured by the number of passes the algorithm must make over the stream, when constrained by the size of available storage space. The model is very appropriate for example when analyzing network traffic data [15,10,12,13]. The amount of data that is generated in such applications is very large, and certainly can exceed any reasonable amount of available memory quickly. Therefore any analysis technique has to assume that it can make only one pass over the data. Recently proposed techniques include clustering algorithms when the objects arrive as a stream [15], computing decision tree classifies when the classification examples arrive as a stream [10], as well as computing histograms and answering range queries when the values arrive in a stream [12,13].

It is reasonable to consider two stream models: the *window* model, where we are only interested in data that arrived recently, within a window W from current time, and the *complete* model, when all data are equally interesting. Both of the models have limitations, since in the first model old data are completely lost, and in the second model accuracy can suffer since the size of available storage remains constant while the amount of data increases continuously.

In this paper we assume that insertions and deletions of tuples come as a stream. Since both the window model and the complete model have disadvantages, we propose a *hybrid* model called the *Hierarchical Temporal Aggregation (HTA) model*. We keep full information of all tuples that arrived during the most recent time, but we aggregate earlier records at coarser granularities. We further separate the HTA model into the *fixed storage model* and the *fixed time window model*. The difference is based on the mechanisms to control the aggregation. The former one is based on the size of the structures, and the latter one is based on the amount of time that passes.

Besides considering the (plain) temporal aggregation, we also consider the more general range-temporal aggregation as well as the spatio-temporal aggregation problem. A range-temporal aggregation query is: "find how many phone calls were made in 1999 from phones in the 626 area (Los Angeles)". An example of a spatio-temporal aggregation query is: "given an arbitrary spatial region, compute the total rain precipitation in this region in 1999".

The contributions of the paper can be summarized as:

- We provide efficient solutions using specialized aggregation indices for temporal aggregate problems under multiple granularities. In particular we propose solutions for both the fixed storage model and for the fixed time window model.
- Based on the temporal aggregation solutions, we propose a general framework of computing aggregates using multiple time granularities.
- We show how the framework can be specialized to solve the range-temporal aggregation and the spatio-temporal aggregation problems under the fixed time window model.
- Finally, we present extensive experimental results that validate the newly proposed structures.

The ability to aggregate with fixed storage is reminiscent of the *on-line aggregation* proposed in [16]. There, when an aggregation query is asked, an approximate answer is given very quickly. This answer is progressively refined, while the user is provided with the confidence of the current answer and the percentage of the elapsed computation time out of the total time needed to get the final answer. [20] achieved a similar goal by keeping aggregate information in the internal nodes of index structures. The problem examined in this paper is different since we do not keep all the information as the existing works do, and we always give exact answers to the aggregation queries very fast, with the exact answers for the remote history being aggregated at coarser time granularities.

Aggregating with different granularities also resembles the *roll-up* operation examined in data warehousing studies. A data warehouse is usually based on a multi-dimensional data model. Along each dimension, a *concept hierarchy* may exist, which defines a sequence of mappings from a set of low-level concepts to high-level, more general concepts. For example, along the *location* dimension, we may have a concept hierarchy $city \in state \in country \in continent$. The roll-up operation performs aggregation by climbing up a concept hierarchy for a dimension. In a sense the HTA model also aims to "roll-up" by climbing up the time hierarchy. However, the problem examined here is different for two reasons: (1) In a data warehouse, the information is stored at the finest granularity, while in the HTA model, information is stored at different time granularity, according to whether the information is old or new; and (2) The roll-up in data warehousing takes place at query time, which allows the user to get the aggregation results at coarser granularity; while in the HTA model, the roll-up takes place automatically and systematically as data accumulates.

Related is also the work in [27], which presents an effective technique for data reduction that handles the gradual change of the data from new detailed data to older, summarized data in a dimensional data warehouse. Although the time dimension can be considered as an ordinary dimension in the data warehouse, our work differs in the semantics of data over the time dimension. Consider the case when a tuple is valid across all days in a week. In our environment, the tuple value will be counted once towards any query which intersects the valid week. In the data warehouse environment, however, the tuple value will be counted once for every day of the week. Thus to aggregate by week, the technique of [27] will multiply the tuple value by 7. Clearly, both semantics have practical meanings. However, they apply to different scenarios.

The rest of the paper is organized as follows. Section 2 presents the models of aggregation and identifies four implementation issues. The solution for the temporal aggregation problem under the fixed storage model is presented in section 3, while the solution under the fixed time window model is presented in section 4. Based on these solutions, we propose a general framework for computing time-related aggregates using multiple time granularities in section 5. As special cases of the framework, the solutions to the range-temporal aggregation problem and the spatio-temporal aggregation problems under fixed time window model are given in sections 6 and 7, respectively. The performance results appear in section 8. Section 9 discusses related work while conclusions are presented in section 10.

2. Problem Definition

The time dimension naturally has a hierarchy. A k-level time hierarchy is denoted as $gran_1 \rightarrow \ldots \rightarrow gran_k$, where $gran_1$ is at the coarsest granularity and $gran_k$ is at the finest granularity. Here each granularity has a value range (normally a finite subset of the set of integers). Any time instant corresponds to a full assignment of the k granularities. For example, a 3-level time hierarchy may be: $year \rightarrow month \rightarrow day$. Here at the finest granularity we have days, which are grouped in months, which are further grouped in years. A time instant "Oct 1, 2001" corresponds to the following full assignment of the three granularities: "$year$=2001, $month$=10, day=1".

Given a k-level time hierarchy, the *Hierarchical Temporal Aggregation (HTA) model* divides the time space $[orig,\ now)$ into k segments and for each segment $i \in \{1, \ldots, k\}$, the aggregates are maintained with granularity $gran_i$ (figure 1). Here $orig$ is a fixed time specifying the creation time of the database and now corresponds to the ever-increasing current time. The *dividing times* between pairs of adjacent segments are denoted as $t^1_{div}, \ldots, t^{k-1}_{div}$.

As now progresses and as new objects are inserted, the initial assignments of the dividing times may become obsolete. Thus, any solution to the temporal aggregation problem under the HTA model should allow the corresponding segments to be dynamically adjusted. Depending on what triggers the dynamic adjustment, we further separate the HTA model into two sub-models:

- **Fixed Storage Model.** The assumption for this model is that the available storage is limited. When the total storage of the aggregate index becomes more than a fixed threshold S, older information is aggregated at a coarser granularity.
- **Fixed Time Window Model.** Here we assume that the lengths of all segments (except the first $segment_1$) are fixed. For example, in $segment_k$ where records are aggregated by $gran_k$ (say by day), we may want to keep information for one year; for $segment_{k-1}$ where records are aggregated by $gran_{k-1}$ (say by month), we may want to keep information for five years, etc. Hence, as now advances, we need to increase the dividing time instants $t^1_{div}, \ldots, t^{k-1}_{div}$.

The two models are geared towards different requirements. For example, in a telephone traffic application we may want to always maintain the aggregate over the most recent day (i.e., the window size of the latest segment, $segment_k$ is set to a day). On the other hand, the fixed storage model guarantees its storage requirements but the length of the aggregation maybe less, equal or more than a day.

To increase a dividing time t^i_{div} in the fixed time window model, both the aggregate indices for $segment_i$ and $segment_{i+1}$ should change. To maintain efficiency the following is needed: (a) We should avoid building the complete index structures for the two segments around t^i_{div} whenever it advances. Rather, some kind of *patching* should be performed. (b) We should avoid frequent dividing time advances. If we actually maintain $length_i$ to be a fixed value, the dividing times should advance every time now advances. Instead, we allow $length_i$ to be a value within a certain range $[W_l, W_h)$ and we increase t^{i-1}_{div} whenever $length_i$ reaches W_h.

To design an aggregation index under any of the two models, there are four issues that need to be addressed.

1. **Structure:** We need to maintain an index for each of the k segments and we need to integrate these indices (and possibly some additional information) together as a unified index structure.

2. **Query:** We need an algorithm to evaluate an aggregation query by looking up the unified index structure. One important issue is

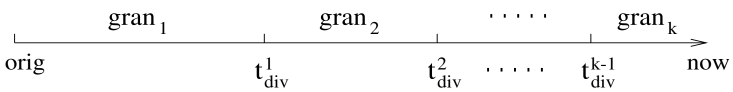


Figure 1. The HTA model.

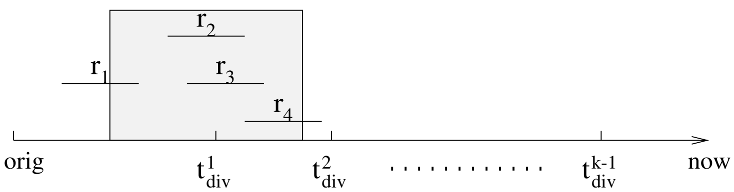


Figure 2. A record may affect several segments.

that for a query which touches several segments, we should avoid counting the same record more than once. In figure 2, the shadowed box illustrates the query, which involves segment 1 and segment 2. The query result is the total weight of records r_1, r_2, r_3, r_4. However, since we maintain the aggregates for each segment individually, record r_2 and r_3 are maintained in both segments. So if the query is divided into two, one per segment and aggregate the two query results, r_2 and r_3 would be counted twice, which should be avoided. Another issue that arise upon query is the query granularity. As figure 2 shows, a query may span more than one segments, where each segment has a different granularity. In this case, we assume the query granularity to be the coarsest one among all these segments.

3. **Update:** We need to incrementally maintain the unified index structure as new objects are inserted/deleted.

4. **Advancing the dividing times:** The two sub-models differ in how the advancement is triggered. However, for both models, we need to advance the dividing times systematically. Moreover, information before the new dividing time should be removed from the index and aggregated at a coarser granularity.

3. Temporal Aggregation with Fixed Storage

[30] proposed two approaches to incrementally maintain temporal aggregates. Both approaches rely on using two *SB-trees* collectively. In this section we first propose a slightly different approach and then we present a technique to extend the approach under the fixed storage model.

3.1. The 2SB-tree

A single SB-tree as proposed in [30] can be used to maintain instantaneous temporal aggregates. One feature about the SB-tree is that a deletion in the base table is treated as an insertion with a negative value. Thus in the rest we focus on the insertion operation. [30] also proposed two approaches to maintain the cumulative temporal aggregate. The first approach is called *Dual SB-tree*. Two SB-trees are kept. One maintains the aggregates of records valid at any given time, while the other maintains the aggregates of records valid strictly before any given time. The latter SB-tree can be implemented via the following technique: whenever a database tuple with interval i is inserted in the base table, an interval is inserted into the SB-tree with start time being $i.end$ and end time being $+\infty$. To compute the aggregation query with query interval i, the approach first computes the aggregate value at $i.end$. It then adds the aggregate value of all records with intervals strictly before $i.end$ and finally subtracts the aggregate value of all records with intervals strictly before $i.start$.

The second approach is called the *JSB-tree*. Logically, two SB-trees are again maintained. One maintains the aggregates of records valid strictly before any given time, while the other maintains the aggregates of records valid strictly after any given time. Physically, the two SB-trees can be combined into one tree, where each record keeps two aggregate values rather than one. To compute an aggregate with query interval i, this approach subtracts from the total value of all maintained intervals (which is a single value and is easily maintained) the aggregate of all records with intervals strictly before $i.start$ and the aggregate of all records with intervals strictly after $i.end$. As [30] points out, the two approaches have tradeoffs and no one of them is obviously better than the other.

We hereby propose a new approach called the *2SB-tree*. The idea is again to maintain two SB-trees. One maintains the aggregates of records whose *start* times are less than any given time, while the other maintains the aggregates of records whose *end* times are less than any given time. To compute a temporal aggregate regarding interval i, we find the total value of records whose *start* times are less than $i.end$ and then subtract the total value of records whose *end* times are less than $i.start$. Note that each such SB-trees takes as input, besides a value, a point instead of an interval. The point implies an interval from it to $+\infty$. Compared with the Dual SB-tree, to answer an aggregation query we need to perform two SB-tree traversals instead of three. Compared with the JSB-tree approach (note that the two trees used in the 2SB-tree approach can also be physically combined into one), there is no need to maintain the total weight of all records, and the two SB-trees have unified point input. In the JSB-tree, we can also let the two SB-trees take point input; however, the input points have different meanings for the two trees. In one tree, it implies an interval from $-\infty$ to the point, while in the other tree, it implies an interval from the point to $+\infty$.

3.2. The 2SB-tree with Fixed Storage

When we have fixed storage space, we extend both SB-trees in the 2SB-tree approach with multiple time granularities under the fixed storage model. Each of the extended SB-trees is called the *SB-tree with fixed storage (SB-tree^{FS})* and the complete index is called the *2SB-tree with fixed storage (2SB-tree^{FS})*. It is enough to discuss the four implementation issues on a single SB-tree^{FS}.

Structure: Similarly with a single SB-tree in the 2SB-tree approach, the SB-tree^{FS} maintains a set of time points and is able to answer the 1-dimensional *dominance-sum query* of the form: "given a time t, find the total value of all points less than t". The extension is that the time dimension is divided into k segments. For each segment, an SB-tree I_i is maintained. The total storage size S can be enforced by requiring that each index I_i occupies no more than S_i disk pages, where $S_1, \ldots, S_k$ can be set by the warehouse manager depending on applications. Furthermore, $\sum_{i=1}^{k} S_i = S$. For simplicity we assume that $S_1 = \ldots = S_k = S/k$. For each dividing time t^i_{div}, a value $Snap_i$ is also kept. This value is the total value of all points inserted before t^i_{div}. Figure 3 illustrates the structure.

Query: In order for the SB-tree^{FS} to compute a dominance-sum, we first find the segment i which contains t and query index I_i. Then, if $i > 1$, $Snap_{i-1}$ is added to the query result. Note that by reducing the temporal aggregation query into dominance-sum queries, we have automatically avoided the problem illustrated in figure 2. This is because in the reduced problem, each input, as well as query, deals not with an interval, but with a point, which falls into exactly one of the segments.

Update: To insert a point with time t and value v, we find the segment i which contains t and insert the point into I_i. Next, v is added to $Snap_i, \ldots, Snap_{k-1}$. Last, if the size of I_i becomes larger than the threshold S/k, we advance the dividing time t^{i-1}_{div}. This reduces the size of I_i by removing part of the index while the removed information is aggregated at $gran_{i-1}$ and stored in index I_{i-1}.

Advancing the dividing time: Figure 4a depicts an example where I_1 is aggregated by month

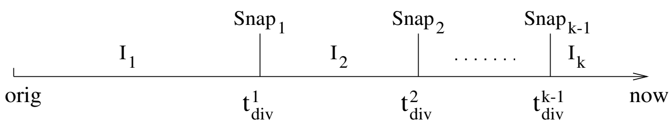
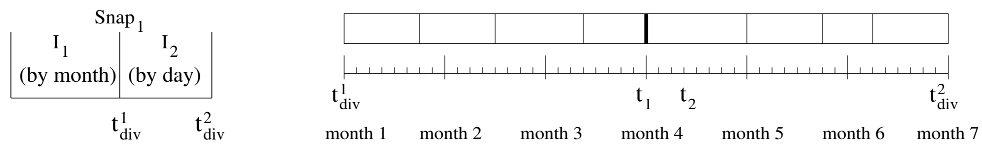


Figure 3. Structure of the SB-tree^{FS}.



(a) The two segments of interest (b) The root-level records of I_2

Figure 4. Advancing the dividing time t^1_{div}.

while I_2 is aggregated by day. Assume I_2 becomes too big and thus t^1_{div} needs to be advanced. Consider the time intervals of the records in the root page of index I_2 (figure 4b). The lifespan of I_2 is between t^1_{div} and t^2_{div}. The major ticks represent the months and the minor ticks represent the days. The dividing time t^1_{div} should occur at a major tick since it is also the end time of the lifespan of index I_1, which is aggregated by month. The rectangles in figure 4b represent the root-level records in SB-tree I_2. The lifespans of these root-level records are contiguous to one another. Since I_2 is aggregated by day, the connection times of these records fall on the minor ticks, but they may not fall on major ticks.

To advance the dividing time t^1_{div}, we need to choose a new dividing time t_{newdiv} and move the information before t_{newdiv} from I_2 to I_1 (aggregated at a coarser granularity). For simplicity let's assume that half of the information is moved over. Straightforwardly, one wishes to go to the root page of I_2 and move the sub-trees pointed to by the left half of the records. For example, in figure 4b, we should move the sub-trees pointed to by the first four root-level records from I_2 to I_1. We first assume that the last root-level record to be moved ends at a major tick (which is the case in the figure, as illustrated by the thick line segment). The case when the record ends somewhere inside a month is discussed afterwards. To remove information from I_2 is easy: we just perform a tree traversal on the sub-trees and deallocate all tree pages met. We thus focus on how to update I_1 and we propose two algorithms.

Algorithm 1. Note that the SB-tree I_2 logically maintains a set of points, each of which corresponds to a month and a day. Since I_1 aggregates by month, we should standardize these points such that they correspond to months only. The algorithm is to find for every month before t_{newdiv}, the total value of points inserted some time during this month. If the value is not zero, we update I_1. For the example of figure 4b, we find the total value of points inserted during months 1, 2 and 3, respectively. To find the value for a given month, say month 2, we perform a dominance-sum query at the first day of month 3 and then subtract from it the dominance-sum at the first day of month 2.

In the above algorithm, the number of queries we perform on I_2 for an advance of dividing time is equal to M, the number of months between t^1_{div} and t_{newdiv}. Thus the total I/O spent for these queries is $O(M \log_B N)$, where N is the number of leaf records in I_2 and B is the page size in number of records. Although M is normally smaller than N, this is not always true; e.g. when the application generates sparse data where each record spans several months and different records have little overlap. We hereby propose a $O(N \log_B N)$

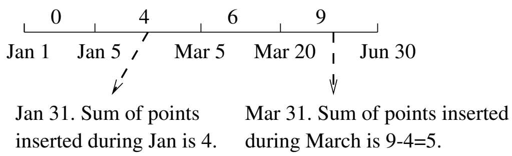


Figure 5. Illustration of algorithm 2.

algorithm. We can choose between the two algorithms at run time when the values of M and N are known.

Algorithm 2. The idea is that we reconstruct the entire aggregates on the part of I_2 that needs to be removed. As shown in [30], this can be performed by a depth-first traversal to reach all leaf nodes. The result is a list of intervals, one connecting to another, each of which is associated with a value (the dominance-sum at all time instants during this interval). An example of such a list is shown in figure 5. We use this example to illustrate how we can identify the months and values to be inserted into I_1 by traversing this list only once. Initially, we meet the first interval [Jan 1, Jan 5). Since it starts and ends in the same month, no change is needed. When we see the second interval [Jan 5, Mar 5), we conclude that in January (the month of the start time of the interval), the sum of all inserted points is 4. This triggers an update of I_1. The next interval [Mar 5, Mar 20) can be ignored again. The last interval [Mar 20, Jun 30) has value 9. The value is the sum of all points inserted in January and in March. To get only the sum of points inserted in March, we subtract 4 from it and we get $9-4=5$. This triggers the other insertion in I_1.

Now we consider the case when the last root-level record to be moved does not end at a major tick. E.g. in figure 4b, assume that the fourth record ends not at t_1, but at t_2. To handle this situation, we choose t_{newdiv} to be the last major tick before t_2 (which is t_1 in this example). There is a single path in I_2 from root to leaf along which each record contains t_1. These records are all split at t_1; the left copies are aggregated and recorded (along with the sub-tree rooted by the first three root-level records) in I_1, while the right copies are retained in I_2.

After the sub-trees are removed from I_2, index I_1 is updated and the dividing time t^1_{div} is advanced to t_{newdiv}, there is one more issue to resolve. Suppose the dominance-sum at t_{newdiv} on the previous I_2 is v, we need to add v to $Snap_1$. Furthermore, we need to update I_2 such that a dominance-sum query within the new I_2 is not affected by the part that were removed. This is achieved by inserting into I_2 value $-v$ at time t_{newdiv}.

Theorem 1 *Let S be the storage limit in number of disk pages and B denote the page capacity in number of records. The 2SB-tree^{FS} with k time granularities computes a temporal aggregate with $O(\log_B(S/k))$ I/Os while the amortized insertion cost is $O(k\log_B(S/k))$.*

Proof. An SB-tree^{FS} with k time granularities contains k SB-trees, one for each granularity. Thus a 2SB-tree^{FS} with k time granularities contains $2k$ SB-trees. We can allocate $S/2k$ pages for each SB-tree. Thus the max number of leaf records in every SB-tree is $O(SB/k)$. Since the query cost for a single SB-tree is $O(\log_B n)$ [30], where n is the number of leaf records, and to compute a temporal aggregate the 2SB-tree^{FS} performs one query in each of the two SB-trees, the query cost is $O(\log_B(SB/k)) = O(\log_B(S/k))$.

Now we consider the insertion cost. We know that the insertion cost of a single SB-tree is $O(\log_B n)$. By applying similar reasoning as in the previous paragraph, we arrive at the conclusion that the insertion cost of the 2SB-tree^{FS} is $O(\log_B(S/k))$ if no advancing of the dividing time takes place. If an SB-tree I_i occupies more than S/k pages due to an insertion, we need to move half of the tree to I_{i-1}. Assume that we use the second algorithm as discussed earlier in this section. The cost contains three parts: (1) the cost to reconstruct the aggregates for the to-be-removed part of I_i; (2) the cost to remove from I_i; and (3) the cost to insert into I_{i-1}. The cost of part 1 is $O(S/k)$ [30]. The cost to remove from I_i is also $O(S/k)$. The cost to insert into I_{i-1} is $O(\frac{SB}{k}\log_B(S/k))$, since there are $O(SB/k)$ intervals in the reconstructed aggregate list and (in

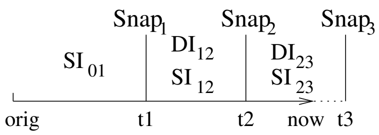


Figure 6. Structure of the SB-tree^{FTW}.

the worst case) for each one of them an insertion takes place in I_{i-1}. In the worst case, the operation of moving half of the tree to another SB-tree (which aggregates at a coarser granularity) is executed recursively for $k-1$ times. So we conclude that the cost of advancing a dividing time is $O(SB \log_B (S/k))$. Now, since after each such advancement, the SB-tree where half of the tree was removed is only half full, to fill it takes another $O(SB/k)$ insertions. Thus the cost of advancing a dividing time can be amortized among the $O(SB/k)$ insertions, or $O(k \log_B (S/k))$ per insertion. To sum up, the amortized insertion cost of the 2SB-tree^{FS} is $O(k \log_B (S/k))$. □

4. Temporal Aggregation with Fixed Time Window

For simplicity, we assume that there are two levels in the time hierarchy: *day* → *minute*. Our solution in this section can be extended to three or more levels. The time dimension $[orig,\ now)$ is divided into two segments by the dividing time t_{div}. As discussed before, the difference between *now* and t_{div} should be within a certain range $[W_l,\ W_h)$. Without lose of generality, we assume that $W_l = W$ and $W_h = 2W$, i.e. $now \in [t_{div} + W, t_{div} + 2W)$. Similar to the methodology of section 3, we extend each of the SB-trees in the 2SB-tree under the fixed time window model (*SB-tree*^{FTW}), and we call the collective structure the *2SB-tree with fixed time window (2SB-tree*^{FTW}*)*. We now focus on a single SB-tree^{FTW}.

Structure: Figure 6 shows the structure. Here we use SI to represent *sparse index*, which means to aggregate by day; we use DI to mean *dense index*, which means to aggregate by minute. The dividing time is currently t_1. For points before t_1, we maintain a sparse index SI_{01}. For points after t_1, we maintain two dense indices DI_{12} and DI_{23}, corresponding to the points before and after $t_2 = t_1 + W$. For points belonging to $[t_1, t_2)$ and points belonging to $[t_2, now)$, we also maintain sparse indices SI_{12} and SI_{23}. These indices will be augmented to index SI_{01} later as time advances. Furthermore, at t_1, t_2 and $t_3 = t_2 + W$, we maintain three values $Snap_1$, $Snap_2$ and $Snap_3$ as the total value of the points before t_1, the total value of the points before t_2 and the total value of the points before t_3, respectively.

Query: The algorithm to compute a dominance-sum regarding time t is as follows:

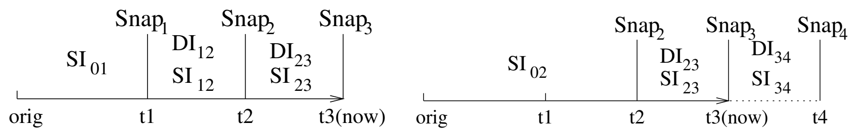
- If $t < t_1$, return the query result on SI_{01};
- otherwise, if $t < t_2$, query SI_{12} or DI_{12}, depending on whether t is the first minute of some day; return the query result plus $Snap_1$;
- otherwise, query DI_{23} or SI_{23}, depending on whether t is the first minute of some day; return the query result plus $Snap_2$;

Update: The algorithm to insert a point with time t and value v is:

- If $t < t_1$, insert into SI_{01}; add v to $Snap_1$, $Snap_2$ and $Snap_3$;
- otherwise, if $t < t_2$, insert into SI_{12} and DI_{12}; add v to $Snap_2$ and $Snap_3$;
- otherwise, insert into SI_{23} and DI_{23}; add v to $Snap_3$.

Advancing the dividing time: When the current time *now* advances to t_3 (figure 7a), we need to advance the dividing time t_{div} from t_1 to t_2. The algorithm is:

- Integrate SI_{12} into SI_{01}. First, reconstruct the aggregates of SI_{12}. Then, scan through the result list of intervals; for each interval $[start, end)$ with value v, insert $\langle start, v - v' \rangle$ into SI_{01}, where v' is the value of the previous interval (for the first interval, v' is 0).



(a) before advancing t_{div} (b) after advancing t_{div}

Figure 7. Advancing the dividing time in the SB-tree^{FTW}.

- Set $Snap_4 = Snap_3$ which stands for the total value of points inserted before $t_4 = t_3 + W$;
- Initialize the dense index DI_{34} and the sparse index SI_{34} to be empty;
- Remove indices DI_{12} and SI_{12}.

After the modifications, the layout of the SB-tree^{FTW} is as shown in figure 7b. We note that the window size W can be adjusted by the warehouse manager. A large W means that the 2SB-tree^{FTW} behaves similar as the 2SB-tree which aggregates only by minute. A small W means that the 2SB-tree^{FTW} behaves similar as the 2SB-tree which aggregates only by day.

5. The General Framework

In sections 3 and 4, we have proposed methods to extend the temporal aggregation index (the SB-tree) to the fixed storage model and to the fixed time window model, respectively. Note that the temporal aggregation problem involves only one dimension: the time dimension. In general, there are time-involved aggregation problems that involve multiple dimensions including the time dimension. Some examples are:

- **range-temporal aggregation:** *Maintain a set of objects, each of which has a key, a time interval and a value, so as to efficiently compute the total value of objects whose keys are in a given range and whose intervals intersect a given time interval.* This problem was introduced in [31]. Compared with the previously discussed temporal aggregation problem, a query is further limited by a key range. The previous problem is actually a special case in that the query key range is always the whole key space. An example scenario is to maintain a database of phone call records, each of which has a key (the telephone number), a time interval during which the call occurred and a value 1. A range-temporal query is: "find how many phone calls were made in 1999 from phones in the 626 area code".
- **spatio-temporal aggregation without spatial extent:** *Maintain a set of objects, each of which has a (d-1)-dimensional point, a time interval and a value, so as to efficiently compute the total value of objects whose points are in a given hyper-rectangle and whose intervals intersect a given time interval.* An example scenario is a database of measured rain falls. Each record has a 2-dimensional point (the location of the measuring instrument), a time interval during which a rainfall occurred, and a value for the measured precipitation. A query may be: "find the total precipitation of rain falls measured in the Los Angeles area in 1999". Note that the above formalization assumes that the query area (e.g. Los Angeles) is a rectangle. Arbitrary query shapes can be divided into a collection of rectangular areas (however details are beyond the scope of this paper).
- **spatio-temporal aggregation with spatial extent:** *Maintain a set of objects, each of which has a (d-1)-dimensional rectangle, a time interval and a value, so as to efficiently compute the total value of objects whose rectangles intersect a given hyper-rectangle and whose intervals intersect a*

> *given time interval.* In the previous rainfall example, although each measuring instrument is located at a *point* location, each rainfall occurs over an area. Again, for simplicity we assume rectangular areas. Thus each record has a 2-dimensional rectangle (the area of the rainfall), a time interval of the rainfall duration, and a value as the measured precipitation.

In this section, we assume the existence of aggregate indices with a single time granularity and we build a general framework of how to extend these structures to the fixed storage model and the fixed time window model. Examples of such single-granularity indices are the SB-tree [30], the MVSB-tree [31] and the BA-tree [32].

As shown in section 3.1, to aggregate objects with 1-dimensional extent (each object has a time interval), we can utilize two SB-trees, each of which maintains a set of point objects (one for the interval start time and the other for the interval end time) and is able to answer the following query: "compute the total value of objects whose points are to the left of a query point". This query is an example of the *dominance-sum* problem. We first define what *dominance* means:

Definition 1 *Given two d-dimensional points* $x = (x_1, \cdots, x_d)$ *and* $y = (y_1, \cdots, y_d)$*, we say that* x dominates y *if for every* $i \in \{1, \cdots, d\}$, $x_i \geq y_i$.

Intuitively, x dominates y if x is located to the upper-right of y. If x dominates y, we equivalently say that y is *dominated by* x.

Definition 2 *Given a collection* S_p *of d-dimensional point objects and a query point p, the* **dominance-sum** *of p regarding* S_p *is SUM{ o.value|o* $\in S_p$ *and o.point is dominated by p}.*

Intuitively, the dominance-sum of a query point p is the total value of objects located to the lower-left of p. For example, in the temporal aggregation case, we used a single SB-tree to compute the total value of objects whose time instants are smaller than a query point (i.e., the 1-dimensional dominance-sum).

Our framework of solving time-related aggregations is as follows. First, the aggregation is reduced to *dominance-sums*. We assume there exist dominance-sum indices with single time granularity. Next we discuss how to extend these dominance-sum structures to support multiple granularities under both HTA models.

5.1. Dominance-Sum Framework with Fixed Storage

For the fixed storage model, i.e. to compute d-dimensional (including the time dimension) dominance-sums with fixed storage space, the extended structure is shown in figure 8. Here the time dimension from *orig* to *now* is divided into k segments, each segment $i \in [1..k]$ corresponding to a single-granularity d-dimensional dominance-sum structure I_i. Again, each such index corresponds to a different granularity, e.g. I_1 aggregates per year, I_2 aggregates per month, and I_3 aggregates per day, etc. At every dividing time t^i_{div} between two adjacent segment indices I_i and I_{i+1}, a (d-1)-dimensional dominance-sum structure (without the time dimension) $Snap_i$ is maintained which is used to pre-compute the (d-1)-dimensional dominance-sums over points to the left of t^i_{div}.

To perform a dominance-sum query, where the query point falls in segment i, first a query is performed on I_i for objects whose times are in segment i. Next, if $i > 1$, a query is performed on $Snap_{i-1}$ for objects whose times are before t^i_{div}. The summation of these queries is the anticipated result.

The total allowed storage space can be allocated to the indices I_i and $Snap_i$. Since the sizes of indices $Snap_i$ are independent to the time dimension, we expect them to be small. So the main focus is the sizes of I_i. As time advances and records accumulate, I_i may become too large. We handle this by advancing the dividing time t^{i-1}_{div} to a later time t_{newdiv}. There are several operations that need to be performed:

1. **divide operation**: divide index I_i into two parts: $I^<_i$ which corresponds to the point objects before t_{newdiv} and $I^>_i$ which corresponds to the point objects after t_{newdiv}. $I^>_i$ becomes the new I_i after the dividing

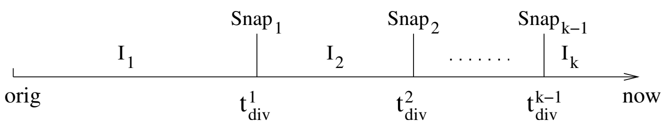


Figure 8. Dominance-sum structure with fixed storage.

time is advanced.

2. **coarsen operation**: coarsen the granularity of $I_i^<$ so that it aggregates at the same granularity of I_{i-1}. The result is usually a much smaller index $I_i^{<'}$.

3. **integrate operation A**: integrate the coarsened index $I_i^<$ with I_{i-1}. The integrated index is the new I_{i-1}.

4. **integrate operation B**: integrate the coarsened index $I_i^<$ with $Snap_{i-1}$. The integrated index is the new $Snap_{i-1}$.

As an example, in section 3.1 we discussed how to extend an SB-tree to multiple granularities. The divide operation is straightforward. As shown in figure 4b, the SB-tree is a B+-tree like structure whose entries correspond to continuous intervals. The sub-tree referenced by the entries before the new dividing time t_{newdiv} is the first sub-tree which is to be coarsened and integrated into the previous index I_{i-1}, while the sub-tree referenced by the rest entries is the new I_i. The coarsen operation and the integration operation A are performed together in section 3.1. The integration operation B is trivial for the SB-tree with fixed storage, since the $Snap_{i-1}$ which is to be updated is a single value (corresponding to a 0-dimensional index structure). The integration simply adds to it the total value of points in $I_i^<$, i.e. those whose time instants are between t_{div}^{i-1} and t_{newdiv}. The value can be located via a single search in the index $I_i^<$.

5.2. Dominance-Sum Framework with Fixed Time Window

If the four operations identified in the previous section are provided by the aggregate indices, the structure shown in figure 8 can also be used for the fixed time window model, too. Since the SB-tree provides all these four operations, the structure of the SB-tree^{FTW} proposed in section 4 is not necessary. However, that discussion illustrates how the framework works when at least one of the four operations is not provided. For instance, this is the case when using the MVSB-tree or the BA-tree for range-temporal and spatio-temporal aggregations, since it is not clear how the divide operation and the integration operations can be performed with these indices. The following discussion assumes that none of the four operations is provided.

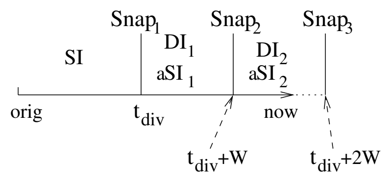


Figure 9. Dominance-sum structure with fixed time window (two granularities).

For simplicity, we consider the case when there are two time granularities. The extended structure is shown in figure 9. Again, the objects whose time instants are before t_{div} are aggregated at a coarser granularity into index SI (standing for *sparse index*), and the objects whose time instants are after t_{div} are aggregated at a finer granularity into index DI_1 and DI_2 (here DI stands for *dense index*). We assume the window size between the dividing time t_{div} and the current time *now* is always between W and $2W$. Thus DI_1 maintains the aggregation of points between t_{div} and $t_{div}+W$, while DI_2 maintains the aggregation of points after $t_{div}+W$. All the indices SI, DI_1 and DI_2 are d-dimensional dominance-sum indices with a single granularity.

When *now* advances to $t_{div} + 2W$, we advance the dividing time to $t_{newdiv} = t_{div} + W$. Since we have already separated the dense index into two parts, there is no need to perform the divide operation. However, we still need to coarsen DI_1 and integrate it into SI. Since we do not assume the dominance-sum index has this oeration, we maintain along with DI_1 a separate index aSI_1, called *append index*, which corresponds to the points between t_{div} and $t_{div} + W$ and which will be integrated into SI when the dividing time is to be advanced. Similarly, we maintain such an index aSI_2 along with DI_2. One straightforward approach to implement such append indices is to index the original objects where the time instants are aligned to coarse granularities. This alignment usually means that the append indices are much smaller than the corresponding dense indices. For example, suppose there are two time granularities *year* $\rightarrow$ *month* and the list of objects are:

- May 1995, value=1
- Sep 1995, value=4
- Oct 1995, value=2
- Feb 1996, value=3
- Apr 1996, value=5
- Aug 1996, value=2
- Sep 1996, value=1
- Nov 1996, value=1

Then the corresponding append index maintains two items:

- 1995, value=7
- 1996, value=12

To support queries, we maintain (d-1)-dimensional indices $Snap_1$, $Snap_2$ and $Snap_3$, which maintain the aggregates of points before t_{div}, points before $t_{div}+W$, and all points, respectively. Note that $Snap_3$ is not necessary, since it is never used in query. The reason for maintaining it is that when we advance the dividing time, rather than computing the new $Snap_2$ (by integrating aSI_2 into the old $Snap_2$), we can simply take the pre-computed $Snap_3$.

As special cases of the framework, in the next two sections we discuss the range temporal aggregation and the spatio-temporal aggregation problems with multiple time granularities. Since neither of the existing corresponding dominance-sum structures (MVSB-tree and BA-tree) provides the four operations pointed out in section 5.1, we focus on the fixed time window model.

6. The Range Temporal Aggregation with Fixed Time Window

We first show how the range temporal aggregation problem can be reduced to dominance-sums in section 6.1. In our earlier work [31], we proposed the MVSB-tree (which we review in appendix A) which is an index structure to maintain dominance-sums under the transaction-time model. In section 6.2 we show how we can extend the MVSB-tree to fixed time window model as a special case of the general framework proposed in the previous section.

6.1. Reduction from Range Temporal Aggregation to Dominance-Sums

In [31], we proposed a technique to reduce each range temporal aggregation query to 6 dominance-sum queries. In this section, we propose a more efficient approach where only 4 dominance-sum queries are needed.

Intuitively, a query time interval and a key range collectively form a query rectangle in the 2-dimensional time-key space. The query rectangle has 4 corners. A range temporal aggregation query is then reduced to 4 dominance-sum queries, one for each corner of the query box. This is illustrated in figure 10. Figure 10a shows a query rectangle and two objects intersecting with it. The range temporal aggregation query asks to compute the total value of these two objects. We first note that in order for an object o to intersect the query box q, the left corner of o has to be dominated by the upper right corner of q. Figure

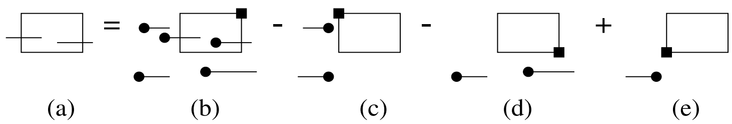


Figure 10. A range temporal aggregation query is reduced to 4 dominance-sum queries

10b shows the candidate boxes. Some candidates are false hits since they are either completely to the left, or completely under, q. The false hits to the left of q correspond to those whose right corners are dominated by the upper left corner of q (figure 10c). The false hits under q correspond to those whose left corners are dominated by the lower right corner of q (figure 10d). Note that after these false hits are subtracted from the query result, the objects whose right corners are dominated by the lower left corner of q (figure 10e) are subtracted twice. So their sum must be added again. To sum up, we can maintain two MVSB-trees, one for the left corners of all objects and the other for the right corners of all objects; and a range temporal aggregation query is reduced to 4 dominance-sum queries in the two MVSB-trees. We call this new structure the *2MVSB-tree*.

6.2. The MVSB-tree with Fixed Time Window

In this section we show how the MVSB-tree can be extended to the fixed time window model. We call such an extended structure the *MVSB-tree*^{FTW}. Due to our reduction technique in section 6.1, we can compute range-temporal aggregates using two MVSB-tree^{FTW}.

Using the two-granularity case as an example, the structure of the MVSB-tree^{FTW} is the same as shown in figure 9. The indices SI, DI_1 and DI_2 are implemented as MVSB-trees. The append indices aSI_1 and aSI_2 are arrays. Again, we expect that these arrays are small for two reasons: (a) the lengths of intervals $[t_1, t_2)$ and $[t_2, t_3)$ should be within some given window sizes; and (b) the arrays are sparse, i.e. if during the same year there are many points inserted with the same key, they are kept as a single array element. The indices $Snap_1$, $Snap_2$ and $Snap_3$ are implemented as SB-trees indexing the key dimension.

The query algorithm and the algorithm to advance dividing times follow the framework precisely and we omit their discussion. There is a minor difference is the update algorithm. While the SB-tree^{FTW} assumes the valid-time model, the MVSB-tree^{FTW} assumes the transaction-time model. Thus any point inserted in the MVSB-tree^{FTW} has time *now*. So to insert a new point, the only indices to be affected are DI_{23}, SI_{23} and $Snap_3$.

7. The Spatio-Temporal Aggregation with Fixed Time Window

In section 7.1 we review our technique which reduces the spatio-temporal aggregation problem to the dominance-sum problems [32]. In the same paper, we proposed the BA-tree which precomputes d-dimensional dominance-sums (the BA-tree is reviewed in appendix B). We show how to extend the BA-tree to solve the spatio-temporal aggregation problem with fixed time window in section 7.2.

7.1. Reduction to Dominance-Sums and the BA-tree

The spatio-temporal aggregation problem WITHOUT spatial extent is a special case of the spatio-temporal aggregation problem WITH spatial extent. Thus in this section we focus on the latter problem. Each object has a (d-1)-dimensional spatial rectangular region and a time interval. In fact, we can treat time dimension as an ordinary dimension and thus the object rectangle and time interval collectively form a d-dimensional hyper-rectangle, also called a *box*. Similarly, a query also has a box. The spatio-temporal aggregation problem is equivalent to the following spatial aggregation problem: “compute the total value of objects whose boxes intersect a given box".

In [32], we called this problem the *box-sum* problem and we proposed a technique to reduce it into the dominance-sum problem. Note that the SB-tree and the MVSB-tree cannot be used to solve the general d-dimensional dominance-sum problem for the following reasons: the SB-tree solves the 1-dimensional dominance-sum problem, but not higher dimensions; the MVSB-tree solves the 2-dimensional dominance-sum problem, but it applies only to the transaction-time model and it cannot address higher dimensions as well. In [32], we proposed the *BA-tree* which computes d-dimensional dominance-sums. Below we review the reduction technique from box-sum to dominance-sum. A description of the BA-tree appears in the Appendix.

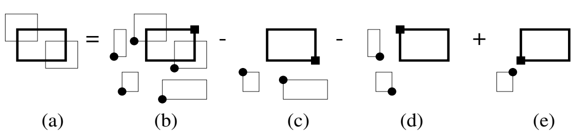


Figure 11. A 2-dimensional box-sum query is reduced to four dominance-sum queries.

The reduction technique is illustrated by figure 11. Intuitively, a d-dimensional box has 2^d corners. An index is maintained for each given corner (e.g. the upper-right corner) of all the objects. A box-sum query is then reduced to 2^d dominance-sum queries, one for each corner of the query box. As an example, consider figure 11 in the 2-dimensional space. Figure 11a shows a query box (with thick border) and two objects intersecting with it. The box-sum query computes the total value of these two objects. In order for a box b to intersect the query box q, the lower left corner of b has to be dominated by the upper right corner of q. Figure 11b shows the candidate boxes. Some candidates are false hits since they are either completely to the left, or completely under, q. The false hits under q correspond to those boxes whose upper left corners are dominated by the lower right corner of q (figure 11c). The false hits to the left of q correspond to those whose lower right corners are dominated by the upper left corner of q (figure 11d). Note that after these false hits are subtracted from the query result, the boxes whose upper right corners are dominated by the lower left corner of q (figure 11e) are subtracted twice. So the total value of them must be added again. The reduction from range-temporal aggregation to dominance-sums as proposed in section 6.1 is actually a special case of the above technique.

7.2. The BA-tree with Fixed Time Window

In this section we show how the BA-tree can be extended to the fixed time window model. We call such an extended structure the *BA-tree*^{FTW}. Since a box-sum query is reduced to several dominance-sum queries, we can compute spatio-temporal aggregates using some BA-tree^{FTW}.

Again, we use the two-granularity case as an example. The structure of the BA-tree^{FTW} is the same as shown in the framework (figure 9). Here, the indices SI, DI_1 and DI_2 are implemented as d-dimensional BA-trees, where one of the d dimensions corresponds to time and the rest dimensions correspond to space. The append indices aSI_1 and aSI_2 are arrays. Note that such arrays are relatively larger than the temporal aggregation case and the range-temporal aggregation case. The reason is that although two points which differ only in time might be combined together, with multiple spatial dimensions it is more likely that two points have different spatial locations and thus cannot be combined. The indices $Snap_1$, $Snap_2$ and $Snap_3$ are implemented as (d-1)-dimensional BA-trees. The algorithms of the BA-tree^{FTW} follow the discussion of the framework.

8. Performance Results

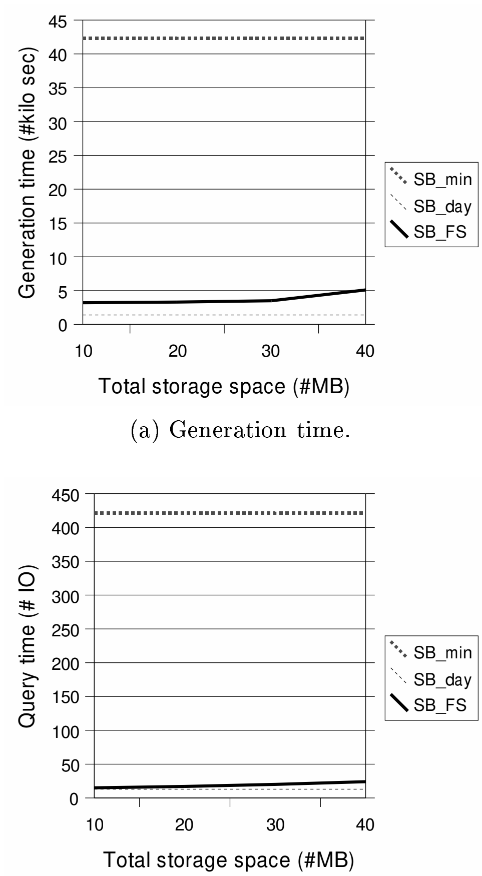
In this section, we provide experimental results to evaluate our framework across various aggregation problems: the temporal aggregation with fixed storage, the temporal aggregation with fixed time window, the range temporal aggregation with fixed time window, and the spatio-temporal aggregation with fixed time window. We compare our proposed solutions with two baseline approaches which involve a single time granularity. Specifically, we assume the time hierarchy has two levels: $day \rightarrow minute$, and consider the following two single-granularity indices, one which aggregates only by day and one that aggregates only by minute. We expect that a by-day aggregation index will have the smallest index size, since it aggregates at a coarse granularity. However, it will also have the least aggregation power in the sense that it cannot compute by-minute aggregates. Conversely, the other baseline case, the by-minute aggregation index, has much higher index size. We will show that the new approaches combine the benefits of the baseline cases.

The algorithms were implemented in C++ using GNU compilers. The programs ran on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.8. We compare the index generation time, the query time and the result index sizes. When generating the indices, we report the time spent in CPU and for I/O. We measure the CPU cost by adding the amounts of time spent in *user* and *system* mode as returned by the *getrusage* system call. We measure the I/O cost by multiplying the number of I/O's by the average disk page read access time (10ms). We used a 8KB page size. For all the algorithms we used the LRU buffering scheme and the buffer size was 100 pages.

Unless otherwise stated, each data set contains 10 million updates. The current time *now* advances gradually until 40 years, which is equal to 14,610 days. For the fixed time window experiments, the average distance between the dividing time and *now* is a parameter in the performance graphs. We call this distance the *by-minute window size* since the records in this *moving window* are aggregated by minute. The by-minute window size varies from 0.1% to 10% of the 40-year time space. To generate the time interval of a record, we choose the end time of its interval to be exponentially close to *now*. The length of the record intervals are on average 1000 minutes.

For the query performance we measure the execution time in number of I/Os of 100 randomly generated queries. For the by-day indices, the aggregates are computed at a by-day granularity. For the by-minute indices, the aggregates are computed at a by-minute granularity. For the indices with multiple granularities, the *recent* aggregates (if the query interval is later than the dividing time) are computed by-minute and the earlier aggregates are computed by-day.

8.1. Performance of Temporal Aggregation with Fixed Storage



(a) Generation time.

(b) Query time.

Figure 12. Generation time and query time of temporal aggregation with fixed storage.

First, we experimented with the 2SB-tree with fixed storage for temporal aggregations. In the figures it is represented as *SB_FS*. As baseline comparison, we also implemented the single granularity 2SB-trees *SB_day* and *SB_min*, that aggregate only by day and only by minute, respectively. For the SB_FS, we vary the storage space from 10MB to 40MB. Figure 12a compares the generation time of the three indices. The SB_min takes much longer to generate than the SB_day. As the amount of space allocated to the SB_FS increases, the SB_FS takes longer to generate, since relatively more records are aggregated by minute, and thus the index size is larger. However, the generation time of SB_FS is always closer to the faster SB_min.

The query time (figure 12b), shows a similar trend, i.e. the SB_FS is very close to SB_day and both are much faster than the SB_min.

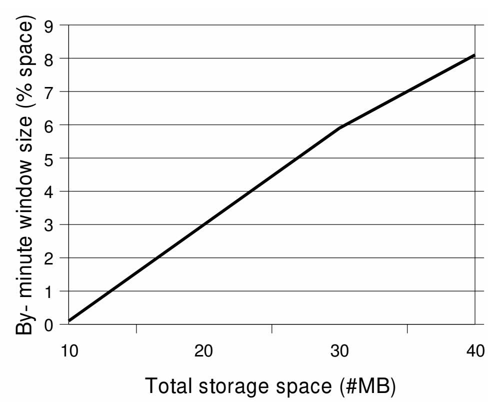
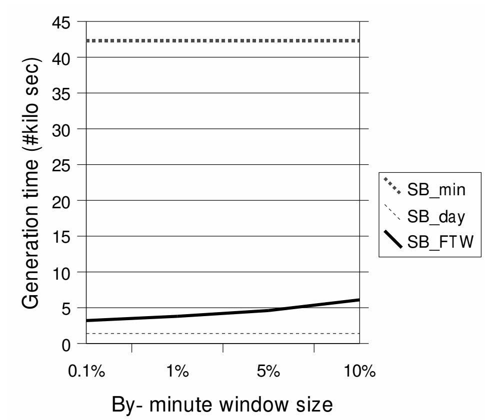


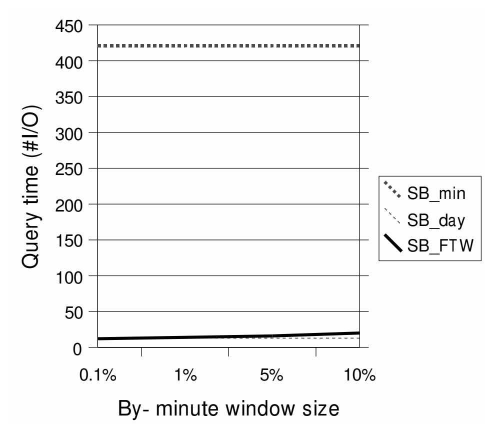
Figure 13. Average by-minute window size of temporal aggregation with fixed storage.

Finally, we examine the relationship between the storage space and the average size of the by-minute window of the SB_FS (figure 13). Note that the 40 year time space is divided into two parts: the first part aggregates by day and the second part aggregates by minute. Obviously, the larger the allocated storage size is, the larger the by-minute window size is. (In the extreme case, when we have unlimited storage, all data is aggregated by minute.) This trend is observed in figure 13.

8.2. Performance of Temporal Aggregation with Fixed Time Window



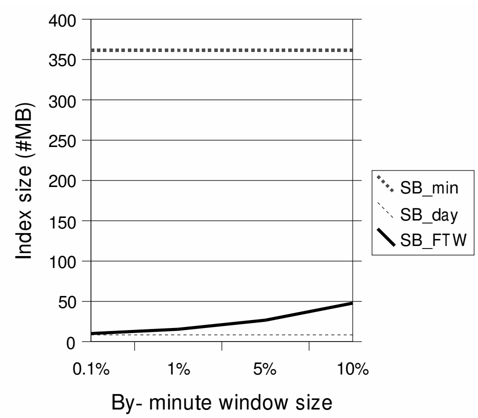
(a) Generation time.



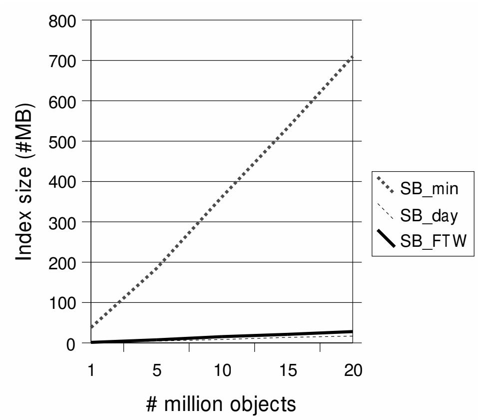
(b) Query time.

Figure 14. Generation time and query time of temporal aggregation with fixed time window.

We use *SB_FTW* to represent the 2SB-tree with fixed time window. Figure 14a compares



(a) Varying window size.



(b) Varying # objects.

Figure 15. Index sizes of temporal aggregation with fixed time window.

the index generation time while varying the by-minute window size. The single-granularity indices SB_day and SB_min are not affected when the by-minute window size varies. As expected, the SB_day takes the shortest time to generate, while the SB_min takes the longest time. The generation time of the SB_FTW is between the other two (11 times less than that of the SB_min for 1% window size). As the by-minute window size becomes larger, the generation time of the SB_FTW tends to be longer, too. The effect of the size of the by-minute window on the generation time is twofold. A larger window size means that the dividing time is increased less often. On the other hand, a larger window implies that the sizes of the indices which exist after the dividing time are larger and thus the updates in them take longer. The combined effect is shown in figure 14a.

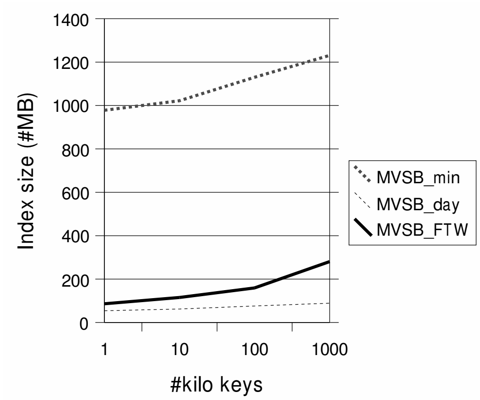
As shown in figure 14b, the SB_FTW and the SB_day have similar query performance which is much faster than that of the SB_min. For 1% window size, the SB_FTW is 30 times faster than the SB_min. The SB_FTW is preferred over the SB_day since for the recent history, the SB_FTW has the ability to aggregate at a finer granularity.

As figure 15a shows, the SB_FTW uses a little more space than the SB_day, but much less than the space used by the SB_min. For 1% window size, the SB_FTW uses 23 times less space than the SB_min. In figure 15b we compare the index sizes where the number of objects changes from 1 million to 20 million. Here we fix the by-minute window size to be 1% of the time space. As the number of objects increases, the sizes of all the three indices increase as well. Still, we observe that the size of the SB_FTW is a lot less than that of the SB_min.

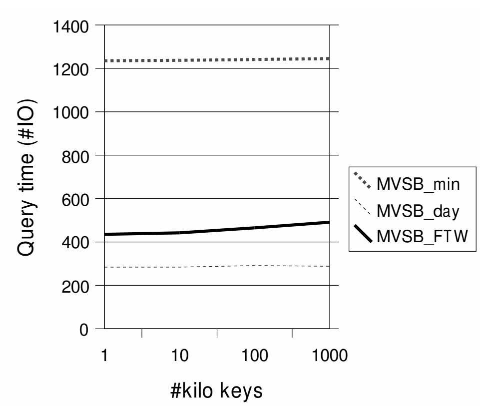
8.3. Performance of Range Temporal Aggregation with Fixed Time Window

We use *MVSB_FTW* to represent our range temporal aggregation index with fixed time window, while *MVSB_day* and *MVSB_min* represent the single-granularity indices. Different from the SB_FTW, now the *Snap* structures are implemented as SB-trees, whose sizes depend on the number of different keys the dataset has. We create four datasets, with the number of keys varying from 1000 to 1,000,000. For the experiments in this section, we use a by-minute window size of 10% of the time space.

Figure 16a shows that the sizes of all three indices increase with the number of different keys. The MVSB_FTW has index sizes closer to the MVSB_day. Figure 16b compares the query time. Again, as the number of keys increases, the MVSB_FTW query time becomes longer. We note that the performance difference between MVSB_FTW and MVSB_day is some-



(a) Index sizes.



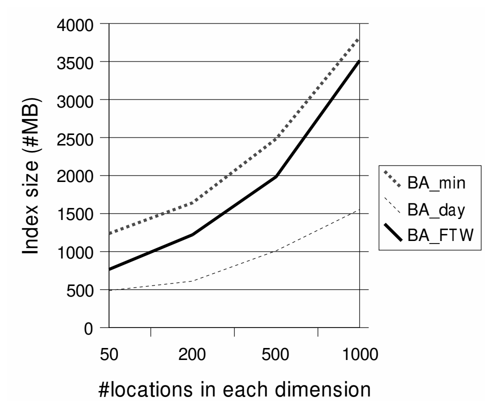
(b) Query time.

Figure 16. Performance of the range temporal aggregation with fixed time window, varying # keys.

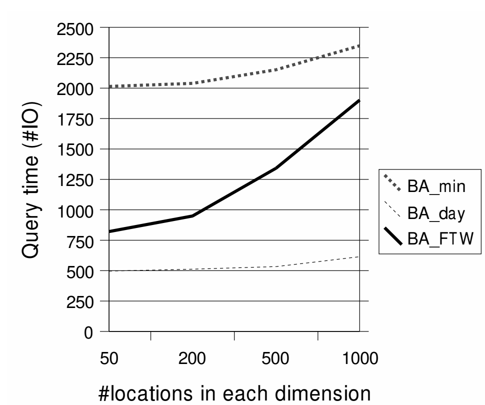
what larger than the performance difference between SB_FTW and SB_day, especially as the number of keys increases. The reason is that in the range temporal aggregation case, the *Snap* structures embedded into the index have sizes proportionally to the number of different keys, and thus storing and querying them incurs extra cost.

8.4. Performance of Spatio-Temporal Aggregation with Fixed Time Window

We use *BA_FTW* to represent our spatio-temporal aggregation index with fixed time window while *BA_day* and *BA_min* denote the corresonding single-granularity indices. For the data sets we generated, besides the time dimension there are two spatial dimensions. The number of different locations per spatial dimension varies from 50 to 1000. Again, the by-minute window size is set to 10% of the time space.



(a) Index sizes.



(b) Query time.

Figure 17. Performance of the spatio-temporal aggregation with fixed time window, varying the number of different locations per spatial dimension.

As shown in figure 17, as the number of locations per spatial dimension increases, both the index sizes and the query time of the indices become larger. The reason is that the BA-tree is sensitive to the number of different spatial locations a point object may reside at. This is more apparent with the BA_FTW since to perform a query, besides the main index, we also query a separate 2-dimensional BA-tree (the *Snap* index) which may be large. This observation illustrates the need to aggregate at coarser granularities both along the time dimension and along other dimensions.

9. Related Work

Temporal Aggregation. [28] presented a non-incremental two-step approach where each step requires a full database scan. First the intervals of the aggregate result tuples are found and then each database tuple updates the values of all result tuples that it affects. This approach computes a temporal aggregate in $\mathrm{O}(mn)$ time, where m is the number of result tuples (at worst, m is $\mathrm{O}(n)$; but in practice it is usually much less than n). Note that this two-step approach can be used to compute range temporal aggregates, however the full database scans make it inefficient. [19] used the *aggregation-tree*, a main-memory tree (based on the segment tree [23]) to incrementally compute instantaneous temporal aggregates. However the structure can become unbalanced which implies $\mathrm{O}(n)$ worst-case time for computing a scalar temporal aggregate. [19] also presented a variant of the aggregation tree, the k-ordered tree, which is based on the k-orderness of the base table; the worst case behavior though remains $\mathrm{O}(n)$. [11,29] introduced parallel extensions to the approach presented in [19]. [22] presented an improvement by considering a balanced tree (based on red-black trees). However, this method is still main-memory resident. Finally, [30] and [31] proposed the SB-tree and the MVSB-tree, respectively, which can be used to compute the scalar and range temporal aggregates. Both of them are disk-based, incrementally maintainable and efficient for queries (logarithmic).

Point Aggregation. The solutions proposed in this paper utilizes existing structures to compute dominance-sums. The computational geometry field possesses much research work on the dominance-sum and the more general geometric range searching problem [21,7,1]. Most solutions are based on the *range tree* proposed by [2]. A variation of the range tree which is used to solve the d-dimensional dominance-sum query is called the *ECDF-tree* [2]. The d-dimensional ECDF-tree with n points occupies $O(n \log_2^{d-1} n)$ space, needs $O(n \log_2^{d-1} n)$ preprocessing time and answers a dominance-sum query in $O(\log_2^d n)$ time. Note that the ECDF-tree is a static and internal-memory structure. For the disk-based, dynamic case, [32] proposed two versions of the *ECDF-B-tree*. [32] also presented the *BA-tree* which combines the benefits of the two ECDF-B-trees.

Time Granularity. A glossary of time granularity concepts appears in [4]. [6] deeply investigates the formal characterization of time granularities. [3] shows a novel way to compress temporal databases. The approach is to exploit the semantics of temporal data regarding how the values evolve over time when considered in terms of different time granularities. [8] considers the mathematical characterization of finite and periodical time granularities and identifies a user-friendly symbolic formalism of it. [9] examines the formalization and utilization of semantic assumptions of temporal data which may involve multiple time granularities.

Data Warehousing. The time hierarchy we used is similar to the concept hierarchy in the data warehousing study. [27] proposed a technique to reduce the storage of data cubes by aggregating older data at coarser granularities. [24] states that one difference between the spatio-temporal OLAP and the traditional OLAP is the lack of predefined hierarchies, since the positions and the ranges of spatio-temporal query windows usually do not confine to pre-defined hierarchies and are not known in advance. [24] presented a spatio-temporal data warehousing framework where the spatial and temporal dimensions are modeled as a combined dimension on the data cube. Data structures are also provided which integrate spatio-temporal indexing with pre-aggregation. [14] presented a framework which supports the expiration of unneeded materialized view tuples. The motivation was that data warehouses collect data into materialized views for analysis and as time evolves, the materialized view occupies too much space and some of the data may no longer be of interest. [26] presented efficient methods to aggregate data in an append-only database.

10. Conclusions

Temporal aggregation have become predominant operators in analyzing time-evolving data. Many applications produce massive temporal data in the form of streams. For such applications the temporal data should be processed (pre-aggregation, etc.) in a single pass. In this paper we examined the problem of computing temporal aggregates over data streams. Furthermore, aggregates are maintained using multiple levels of temporal granularities: older data is aggregated using coarser granularities while more recent data is aggregated with finer detail. We presented two models of operation. In the fixed storage model it is assumed that the available storage is limited. The fixed time window model guarantees the length of every aggregation granularity. For both models we presented specialized indexing schemes for dynamically and progressively maintaining temporal aggregates. An advantage of our approach is that the levels of granularity as well as their corresponding index sizes and validity lengths can be dynamically adjusted. This provides a useful trade-off between aggregation detail and storage space. Based on our temporal aggregation work, we summarized a framework for computing aggregates over time-evolving data and we discussed how the solutions can be extended to solve the more general range temporal and spatio-temporal aggregation problems under the fixed time window model. Finally, an extended performance evaluation validated the advantages of the proposed structures. As future work we plan to further investigate techniques to aggregate at multiple spatial granularities as well. Moreover, we are extending our framework to the multiple data stream environment.

REFERENCES

1. P. Agarwal and J. Erickson, "Geometric Range Searching and Its Relatives", *Advances in Discrete and Computational Geometry*, B. Chazelle, E. Goodman and R. Pollack (ed.), American Mathematical Society, Providence, 1998.
2. J. L. Bentley, "Multidimensional Divide-and-Conquer", *Communications of the ACM* 23(4), 1980.
3. C. Bettini, "Semantic Compression of Temporal Data", *Proc. of WAIM*, 2001.
4. C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang, "A Glossary of Time Granularity Concepts", O. Etzion, S. Jajodia and S. M. Sripada (eds.), *Temporal Databases: Research and Practice*, LNCS 1399, 1998.
5. B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), 1996.
6. C. Bettini, S. Jajodia and X. S. Wang, *Time Granularities in Databases, Data Mining, and Temporal Reasoning*, Springer, 2000.
7. M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag Berlin Heidelberg, Germany, ISBN 3-540-61270-X, 1997.
8. C. Bettini and R. De Sibi, "Symbolic Representation of User-Defined Time Granularities", *Proc. of TIME*, 1999.
9. C. Bettini, X. S. Wang and S. Jajodia, "Temporal Semantic Assumptions and Their Use in Databases", *IEEE TKDE* 10(2), 1998.
10. P. Domingos and G. Hulten, "Mining high-speed data streams", *Proc. of SIGKDD*, 2000.
11. J. Gendrano, B. Huang, J. Rodrigue, B. Moon and R. Snodgrass, "Parallel Algorithms for Computing Temporal Aggregates", *Proc. of ICDE*, 1999.
12. J. Gehrke, F. Korn and D. Srivastava, "On Computing Correlated Aggregates over Continual Data Streams", *Proc. of SIGMOD*, 2001.
13. S. Guha, N. Koudas and K. Shim, "Data-Streams and Histograms", *Proc. of STOC*, 2001.
14. H. Garcia-Molina, W. Labio and J. Yang, "Expiring Data in a Warehouse", *Proc. of VLDB*, 1998.
15. S. Guha, N. Mishra, R. Motwani and L. O'Callaghan, "Clustering Data Streams", *Proc. of FOCS*, 2000.
16. J. Hellerstein, P. Haas and H. Wang, "Online Aggregation", *Proc. of SIGMOD*, 1997.

17. M. R. Henziger, P. Raghavan and S. Rajagopalan, "Computing on Data Streams", *TechReport 1998-011, DEC*, May 1998.
18. H. V. Jagadish, I. S. Mumick and A. Silberschatz, "View maintenance issues for the chronicle data model", *Proc. of PODS*, 1995.
19. N. Kline and R. Snodgrass, "Computing Temporal Aggregates", *Proc. of ICDE*, 1995.
20. I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *Proc. of SIGMOD*, 2001.
21. J. Matoušek, "Geometric Range Searching", *Computing Surveys* 26(4), 1994.
22. B. Moon, I. Lopez and V. Immanuel, "Scalable Algorithms for Large Temporal Aggregation", *Proc. of ICDE*, 2000.
23. F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin/Heidelberg, Germany, 1985.
24. D. Papadias, Y. Tao, P. Kalnis and J. Zhang, "Indexing Spatio-Temporal Data Warehouses", *Proc. of ICDE*, 2002.
25. J. Robinson, "The K-D-B Tree", *Proc. of SIGMOD*, 1981.
26. M. Riedewald, D. Agrawal, A. El Abbadi, "Efficient Integration and Aggregation of Historical Information", *Proc. of SIGMOD*, 2002.
27. J. Skyt, C. S. Jensen and T. B. Pedersen, "Specification-Based Data Reduction in Dimensional Data Warehouses", *TimeCenter TechReport* TR-61, 2001.
28. P. Tuma, "Implementing Historical Aggregates in TempIS", *Master's thesis*, Wayne State University, Michigan, 1992.
29. X. Ye and J. Keane, "Processing temporal aggregates in parallel", *Proc. of Int. Conf. on Systems, Man, and Cybernetics*, 1997.
30. J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", *Proc. of ICDE*, 2001.
31. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", *Proc. of PODS*, 2001.
32. D. Zhang, V. J. Tsotras and D. Gunopulos, "Efficient Aggregation over Objects with Extent", *Proc. of PODS*, 2002.

A. Review of the MVSB-tree

In order to support the dominance-sum query in the two-dimensional key-time space, [31] proposed the MVSB-tree. The tree logically maintains a value for every point in the space. For each object with key k, time t and value v, the insertion operation of the MVSB-tree adds v to the values associated with all the points in the rectangle $[k, maxkey)$ x $[t, maxtime)$. Here $maxkey$ and $maxtime$ collectively form the upper-right corner of the complete space. By doing so, a dominance-sum is transformed to a point query: "given key k and time t, find the value associated with this point in the key-time space". Note that it is impractical to actually keep a value for all points in space. Instead, if with a rectangle, all points have the same value, we keep in the index such a rectangle and the value.

In figure 18 which we borrowed from [31], we use examples to illustrate the idea of the MVSB-tree. We assume each index as well as leaf page holds up to six records. Initially, the MVSB-tree has one root page, R_1, which is also a leaf. There is one record in it having $value = 0$ (figure 18a). After we insert a point with key=20, time=2 and value=1, which we represent as $\langle 20, 2\rangle : 1$, the record is split (figure 18b). If a query point dominates the newly inserted point, then we know that the query result should be 1. On the other hand, the query algorithm examines page R_1 and locates the record whose rectangle contains the query point, in this case the upper right record. Since the value of this record is 1, the query algorithm gives correct result. If, instead, the query point does not dominate the newly inserted point, it would be in the region of either in the left record or the bottom-right record, and the algorithm returns 0.

To insert $\langle 10, 3\rangle : 1$ (figure 18c), both the two records on the right should be split. To ensure minimum update cost, we require that an insertion causes at most one split in a page. As shown in the figure, only the record whose region contains the inserted point is split. To ensure the correctness of query algorithm, a query adds up the values of all records below a query point. For example, if in figure 18c, a query is performed

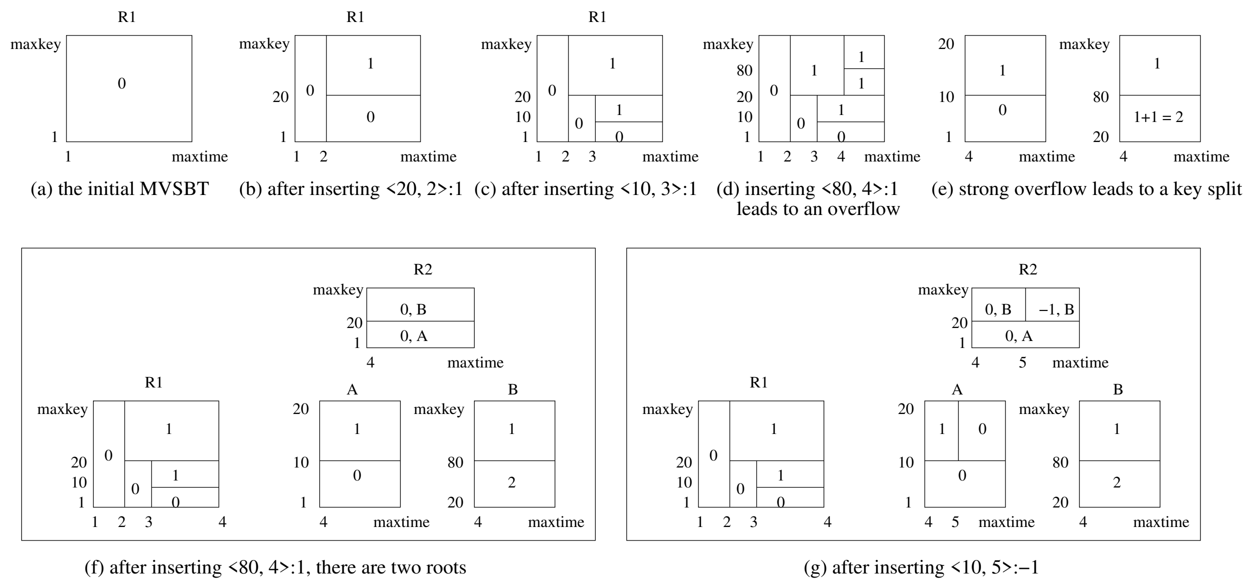


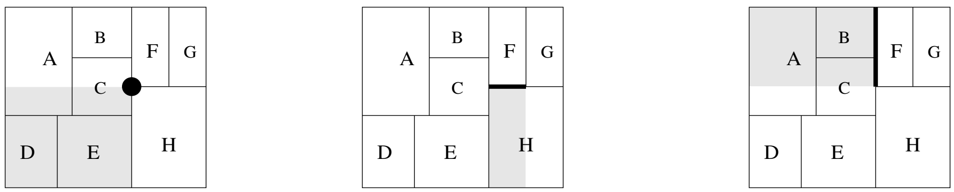
Figure 18. Illustration of an MVSB-tree.

at point $\langle 30, 4\rangle$, we know the query result should be 2, since the query point dominates both inserted objects with value=1 each. If we draw a vertical line segment from the query point down to the bottom of space and add up all the values of records the line segment intersects, we get a result of 2: the correct query result.

The insertion of $\langle 80, 4\rangle : 1$ causes an overflow (figure 18d). A *time split* copies all the alive records (i.e. those whose right borders are *maxtime*) into a new page. To avoid extensive splits, we require a *strong condition* meaning that the number of records in a newly generated page should be within a given range, between 2 and 3 in this example. If the new page satisfied the strong condition, it would be registered as the new root and the insertion would be complete. Note that the MVSB-tree may have multiple root nodes, each one of which corresponds to a time interval and the intervals of adjacent root nodes "connect" with each other. However, in this case the new page strong overflows. So a key split takes place which distributes the records evenly into two pages (figure 18e). Note how the value of the lowest record in the page with higher key range is modified: the total value of all records in the lower-key-range page should be added to the value of it. The tree after the insertion is shown in figure 18f.

We now consider the insertion of $\langle 10, 5\rangle : -1$. In the alive root R_2, the top record is completely above the insertion point, and thus we split it into two, without modifying anything in the sub-tree B it points to (figure 18g). However, this update logically affects all records in B, since a query starts with the root down to the leaf, adding up the values of all records at different levels whose regions contain the query point. The other effect of the update is that for the bottom record which contains the insertion point, the insertion recursively goes to the sub-tree A pointed by it. The overall result after these updates is shown in figure 18g.

Let n be the number of insertions and K be the number of different keys ever inserted; [31] showed that the MVSB-tree answers dominance sum queries in $O(\log_B n)$ I/O's, with $O(\log_B K)$ insertion time and $O(\frac{n}{B} \cdot \log_B K)$ space.



(a) points affecting the subtotal of F (b) points affecting the x−border of F (c) points affecting the y−border of F

Figure 19. The BA-tree is a k-d-B-tree with augmented border information.

B. Review of the BA-tree

In [32] we proposed the BA-tree (*Box-Aggregation Tree*) to compute d-dimensional dominance-sums. It is an extended k-d-B-tree, where the k-d-B-tree proposed by [25] is an index structure which maintains a set of point objects and can efficiently find the points inside a given region. The key idea is that points adjacent to each other are clustered together into disk pages. These disk pages are called leaf pages since they resides at the leaf level of the tree structure. Each leaf page corresponds to a region is space and it is guaranteed that the region of a leaf page contains all points stored in the page. At a higher level, references to adjacent leaf pages are clustered into index pages. Each index page also corresponds to a region which contains the region of all leaf pages it references. If there are multiple index pages, we have yet another level of index pages, and so on. At the highest level, the k-d-B-tree has a single root.

Figure 19a shows the layout of the root node of some k-d-B-tree. Each record (A through H) points to some sub-tree. The k-d-B-tree can be used to compute dominance-sums. Suppose there is a query point p contained in the box of record F (not shown in the figure). In order to compute the total value of points dominated by p, we perform a *range query* where the query box is formed as follows: the lower-left corner of the box is the lower-left corner of the space, while the upper-right corner of the box is p. At the root level, this query box intersects A, C, D, E, H, and possibly B, and thus all these sub-trees are examined. When we reach the leaf level and objects in the query box are located, their values are aggregated on the fly.

This approach is not efficient, since at each level we check multiple child nodes and thus the query performance is linear to the number of objects. This motivates the BA-tree, which achieves poly-logarithmic query time by associating with each index record some *border* information. The objects that may affect the dominance-sum query of a query point in *F.box* (figure 19a) can be divided into four groups: (1) the points contained in *F.box*; (2) the points dominated by the low point of F (in the shadowed region of figure 19a); (3) the points below the lower edge of *F.box* (figure 19b); and (4) the points to the left of the left edge of *F.box* (figure 19c). We can compute the total values of these four groups separately and add up their values to get the final result.

To compute the dominance-sum for points in the first group, a recursive traversal of sub-tree rooted by F is performed. For points in the second group, we keep along with index record F a single value called *subtotal*, which is the total value of all these points. Notice that the subtotal value is independent to where the query point is, as long as it is in F. For computing the dominance-sum in the third group, we can keep an *x-border* in F which contains the x positions and values of all these points. This dominance-sum is then reduced to a 1-dimensional dominance-sum query for the border. It is then sufficient to maintain these x positions in a 1-dimensional BA-tree. Here we can use an SB-tree as the 1-dimensional BA-tree. Similarly, for the points in the fourth group, we keep a *y-border* which is a 1-dimensional BA-tree for the y positions of the group's points.

To summarize, the d-dimensional BA-tree is a k-d-B-tree where each index record is augmented with a single value *subtotal* and two (d-1)-dimensional BA-trees called *x-border* and *y-border*, respectively. The computation for a dominance-sum query at point p starts at the root page R. If R is an index node, it locates the record r in R whose box contains p. The x-border and y-border stored with r and the subtree rooted by r are queries. The final query result is the sum of these three query results plus $r.subtotal$.