Temporal Aggregation over Data Streams using Multiple Granularities

Donghui Zhang* Vassilis J. Tsotras† Dimitrios Gunopulos‡ Bernhard Seeger§

Abstract

Temporal aggregation is an important but costly operation for applications that maintain time-evolving data (data warehouses, temporal databases, etc.). In this paper we examine the problem of computing temporal aggregates over data streams. Such 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 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. Moreover, we discuss how the indexing schemes can be extended to solve the more general range temporal and spatio-temporal aggregation problems.

1 Introduction

With the rapid increase of historical data in data warehouses, temporal aggregates have become predominant operators for data analysis. Computing temporal aggregates is a significantly more intricate problem than traditional aggregation. 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

*Computer Science Department, University of California, Riverside, CA 92521. donghui@cs.ucr.edu

†Computer Science Department, University of California, Riverside, CA 92521. tsotras@cs.ucr.edu. This work was partially supported by NSF grants IIS-9907477, EIA-9983445, and the Department of Defense.

‡Computer Science Department, University of California, Riverside, CA 92521. dg@cs.ucr.edu. This work was partially supported by NSF CAREER Award 9984729, NSF IIS-9907477, the DoD and a gift from AT&T.

§Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany. seeger@Mathematik.Uni-Marburg.de

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 ([Tum92, KS95, YK97, GHR+99, MLI00, YW01, ZMT+01]). 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 ([YW01, ZMT+01]). 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 than the actual data) as well as much faster query times (an adhoc aggregate is computed by simply traversing a path in the index). In particular, [YW01] 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. [ZMT+01] 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. 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 17:30:01 yesterday and 5:30:59 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, [JMS95] 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 [HRR98]. 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 ([GMM+00, DH00, GKS01, GKS01b]). 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 [GMM+00], computing decision tree classifies when the classification examples arrive as a stream [DH00], as well as computing histograms and answering range queries when the values arrive in a stream [GKS01, GKS01b].

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 temporal aggregation problem, we also consider the more general range temporal aggregation problem and the spatio-temporal aggregation problem. An range temporal aggregation query may be: "find how many phone calls were made in 1999 from phones in the 626 area (Los Angeles)". A spatio-temporal aggregation query may be: "find the total precipitation of rain falls measured in the Los Angeles area in 1999".

The contributions of the paper can be summarized as:

- We provide efficient solutions using specialized aggregation indices for the temporal aggregate problems under multiple granularities. In particular we propose solutions for both the fixed storage model and for the fixed time window model. Analytical and experimental results prove the efficiency of our solutions.

- We show how the proposed solutions can be extended to solve the range temporal aggregation and the spatio-temporal aggregation problems under the fixed time window model.

- Furthermore, the proposed specialized index structures can be parameterized. That is, the levels of granularity as well as the corresponding index sizes or validity lengths can be dynamically adjusted. This provides a useful trade-off between the aggregation detail and the storage space.

The ability to aggregate with fixed storage is reminiscent of the *on-line aggregation* proposed in [HHW97]. 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. [LM01] 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 the 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.

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. The performance results appear in section 5. The solutions are extended to solve the range temporal aggregation problem and the spatio-temporal aggregation problems in section 6. Section 7 discusses related work. Finally, section 8 presents our conclusions.

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".

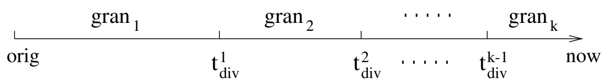


Figure 1: The HTA model.

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.

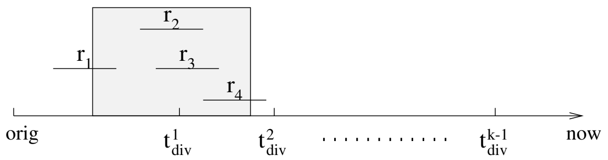


Figure 2: A record may affect several segments.

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

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

[YW01] 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 [YW01] 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. [YW01] 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 [YW01] 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}.

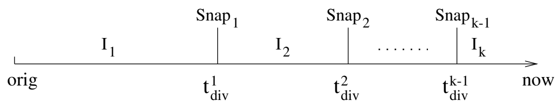
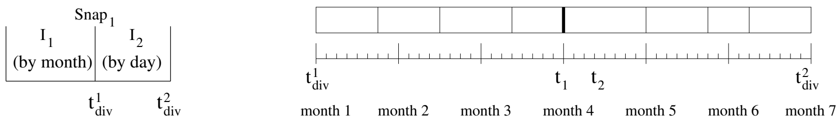


Figure 3: Structure of the SB-tree^{FS}.

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}.



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

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

Advancing the dividing time: Figure 4a depicts an example where I_1 is aggregated by month 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 $O(N \log_B N)$ algorithm. We can choose between the two algorithms at run time when the values of M and N are known.

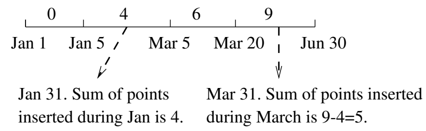


Figure 5: Illustration of algorithm 2.

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 [YW01], 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 *Given a storage limit S, expressed in number of disk pages, the complexities of the 2SB-tree^{FS} with k time granularities are as follows. To compute a temporal aggregate takes $O(\log_B(S/k))$ I/Os, and the amortized insertion cost is $O(k \log_B(S/k))$. Here B is the page* *capacity in number of records.*

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 $O(\log_B n)$ ([YW01]), 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 $O(S/k)$ ([YW01]). 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 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* $\rightarrow$ *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

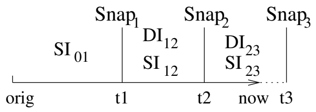


Figure 6: Structure of the SB-tree^{FTW}.

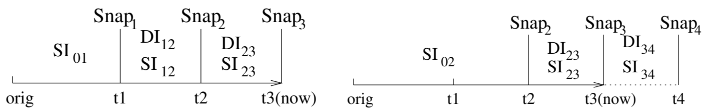
minute. The dividing time is currently t_1. For points before t_1, we maintain a sparse index SI_{01}. For points after t_1, we maintain two dense indices DI_{12} and DI_{23}, corresponding to the points before and after $t_2 = t_1 + W$. For points belonging to $[t_1, t_2)$ and points belonging to $[t_2, now)$, 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 $Snap_3$.



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

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

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).

- 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}.

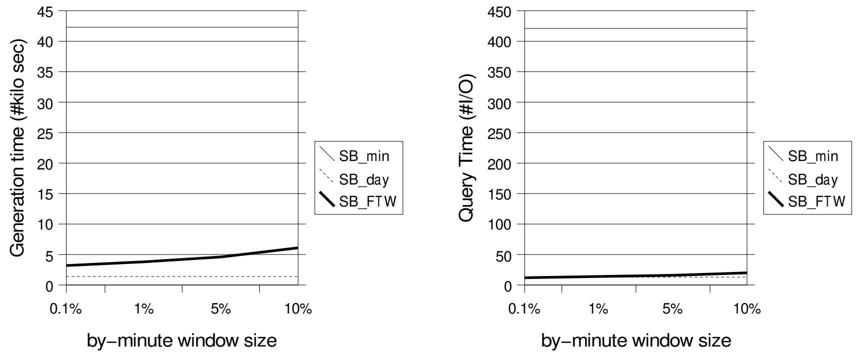
After the modifications, the layout of the SB-tree^{FTW} is as shown in figure 7b. We note that the window size W can be adjusted by the warehouse manager. A large W means that the 2SB-tree^{FTW} behaves similar as the 2SB-tree which aggregates only by minute. A small W means that the 2SB-tree^{FTW}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

5 Performance Results

While the efficiency of the 2SB-tree^{FS} is shown in theorem 1, we do not have a similar theorem for the 2SB-tree^{FTW}. Thus we use performance results to show its efficiency. As baseline cases, we compare with the approaches which involve a single time granularity. Specifically, we assume the time hierarchy has two levels: $day \rightarrow minute$, and we compare our 2SB-tree^{FTW} (denoted as *SB_FTW*) with the 2SB-tree which aggregates only by day (denoted as *SB_day*) and the 2SB-tree which aggregates only by minute (denoted as *SB_min*). We expect that the SB_day uses the least space, since it aggregates at a coarse granularity; however, at the same time it has the least aggregation power in the sense that it does not possess the ability to compute by-minute aggregates. The SB_min is exactly the opposite. We expect that the new approach combines the benefits of both baseline approaches.

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, the CPU cost is an non-trivial part of the total time. So we report the combined generation 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.

Except figure 9b, the data set we used contains 10 million updates. The current time *now* advances gradually by 40 years, which is equal to 14,610 days. 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 a record, we choose the end time of its interval to be exponentially near to *now*. The length of the record intervals are on average 1000 minutes.



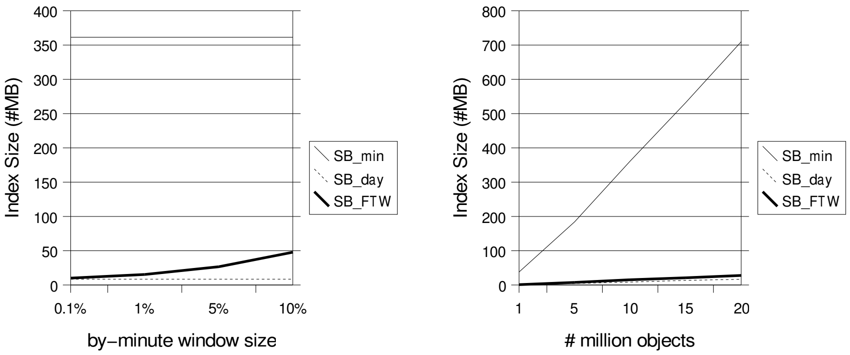
(a) Generation time (b) Query time

Figure 8: Comparing the generation time and the query time.

Figure 8a compares 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. On the one hand, a larger window size means that the dividing time is increased less often. On the other hand, however, a larger window size means that the sizes of the indices which exist after the dividing time are larger and thus the updates in them takes longer. The combined effect is as shown in figure 8a.

For the query performance we measured the execution time in number of I/Os of 100 randomly generated queries. For the SB_day, the aggregates are computed at a by-day granularity. For the SB_min, the aggregates are computed at a by-minute granularity. For the SB_FTW, 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. As shown in figure 8b, the SB_FTW and the SB_day have similar query performance which is much better 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 power to aggregate at a finer granularity.

We now examine the index sizes. As figure 9a 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 9b 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



(a) Varying window size. (b) Varying # objects.

Figure 9: Comparing the index sizes.

than that of the SB_min.

6 The Range Temporal and the Spatio-Temporal Aggregations

We now provide solutions to the range temporal aggregation and the spatio-temporal aggregation problems under the fixed time window model. We first describe the two problems in their original form without the fixed time window model. We then discuss, based on the solutions to the original problems, how to apply the fixed time window model. We do not discuss the solutions to these problems under the fixed storage model since unlike the SB-tree, the existing structures which we can utilize here do not support the operation to remove part of the tree, which is a crucial operation under the fixed storage model.

- **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 [ZMT+01]. Compared with the previously discussed temporal aggregation problem, a query is further limited by a key range. The previous problem is 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 as the telephone number, a time interval during which the call occurred and a value 1. A query may be: "find how many phone calls were made in 1999 from phones in the 626 area (Los Angeles)".

- **spatio-temporal aggregation:** *Maintain a set of objects, each of which has a d-dimensional point, a time interval and a value, so as to efficiently compute the total value of objects whose points are in a given d-dimensional rectangle and whose intervals intersect a given time interval.* An example scenario is to maintain a database of measured rain falls. Each record has a 2-dimensional point as the location of the measuring instrument, a time interval during which a rainfall occurred, and a value as the measured precipitation. A query may be: "find the total precipitation of rain falls measured in the Los Angeles area in 1999". Note that as formalized in the problem definition, we assume the query area (e.g. Los Angeles) is a rectangle. To handle arbitrary query areas, we may break it into a set of rectangular areas. The details are beyond the scope of this paper.

6.1 The Range Temporal Aggregation with Fixed Time Window

The MVSB-tree ([ZMT+01]) has the ability to maintain a set of weighted points in the 2-dimensional time-key space and answer dominance-sum queries of the form: "given point p in the time-key space, find the total value of points dominated by p". Here point p_1 is *dominated* by point p_2 if p_1 has a smaller key and a smaller time than p_2. Due to the transaction-time model, the points have to be inserted in non-decreasing time order. [ZMT+01] proposed to use two MVSB-trees collectively to solve the range temporal aggregation problem. Specifically, each range temporal aggregation query is reduced to 6 dominance-sum queries for the two trees. In this section we first propose a more efficient approach where only 4 dominance-sum queries are needed. We then discuss how to apply the fixed time window model.

6.1.1 The 2MVSB-tree

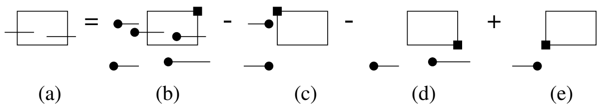


Figure 10: A range temporal aggregation query is reduced to 4 dominance-sum queries

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 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*.

6.1.2 The 2MVSB-tree with Fixed Time Window

The way we extend the 2MVSB-tree under the fixed time window is similar to how the 2SB-tree^{FTW} extends the 2SB-tree. We call the new structure the *2MVSB-tree*^{FTW}, which integrates two *MVSB-tree*^{FTW}. We now discuss in more detail the MVSB-tree^{FTW}.

Structure: The MVSB-tree^{FTW} still has the structure of figure 6. Besides the fact that the indices SI_{01}, DI_{12} and DI_{23} are implemented as MVSB-trees rather than SB-trees, there are two differences between the MVSB-tree^{FTW} and the SB-tree^{FTW}. First, $Snap_1$, $Snap_2$, $Snap_3$ which correspond to t_1, t_2, t_3 are no longer single values, but SB-trees indexing the key dimension. The reason is that the query has a key range, and thus for points before one of these three time instants, say t_1, it is not enough to know the total value. Instead, we need to know the sum of values of the points with keys in a given range. This can be achieved if $Snap_1$ is replaced by an SB-tree indexing the key dimension. Note that the sizes for such SB-trees are independent to the number of points inserted, but is dependent only on the number of different keys. Second, the sparse indices SI_{12} and SI_{23} are not maintained as MVSB-trees, but as 1-dimensional arrays. The reason is that these indices will eventually be integrated into the MVSB-tree SI_{01}, and we cannot integrate an MVSB-tree into another. So we keep SI_{12} and SI_{23} as the original points (at a coarser granularity) which are inserted. Note that the sizes of the arrays are not too big 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 25 points inserted with the same key, they are kept as a single array element.

Query: The query procedure for the MVSB-tree^{FTW} is similar to that of the SB-tree^{FTW} with two differences. First, if the query time is after t_1, while the SB-tree^{FTW} may choose to query either the sparse index or the dense index, the MVSB-tree^{FTW} always queries the dense index. This is because the sparse index here is an array and thus is not efficient to support queries. Second, when the query time is after t_1, while the SB-tree^{FTW} takes the value of $Snap_1$ (or $Snap_2$) and adds it to the answer, we now need to query the SB-tree $Snap_1$ and add the query result to the answer.

Update: The update procedure is different from that of the SB-tree^{FTW}. 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, we always update the indices DI_{23}, SI_{23} and $Snap_3$.

Increase the dividing time: The algorithm is similar to that of the SB-tree^{FTW} with the difference that we need to maintain SB-trees $Snap_1$, etc., rather than single values.

6.2 The Spatio-Temporal Aggregation with Fixed Time Window

As long as we have solutions for the spatio-temporal aggregation problem, we can extend it under the fixed time window model similar to how the 2SB-tree^{FTW} extends the 2SB-tree. In fact, by extending the reduction technique as illustrated in figure 10, we can reduce a spatio-temporal aggregation query into a set of $(d+1)$-dimensional dominance-sum queries. Thus, to solve the spatio-temporal aggregation problem with fixed time window, we need (figure 6) a collection of three $(d+1)$-dimensional dominance-sum structures (SI_{01}, DI_{12}, DI_{23}), three d-dimensional dominance-sum structures (Snap_1, Snap_2, Snap_3) and two evolution lists (SI_{12}, SI_{23}). In the literature, there exist both main-memory solutions (the *ECDF-tree* which was proposed by [Ben80]) and external-memory solutions (the *BA-tree* which was proposed by [ZTG+01]) for the d-dimensional dominance-sum problem. So the spatio-temporal aggregation problem under fixed time window model is solved.

7 Related Work

Temporal Aggregation. [Tum92] 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 $O(mn)$ time, where m is the number of result tuples (at worst, m is $O(n)$; but in practice it is usually much less than n). Note that this two-step approach can be used to compute range temporal aggregates, however the full database scans make it inefficient. [KS95] used the *aggregation-tree*, a main-memory tree (based on the segment tree [PS85]) to incrementally compute instantaneous temporal aggregates. However the structure can become unbalanced which implies $O(n)$ worst-case time for computing a scalar temporal aggregate. [KS95] 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 $O(n)$. [GHR+99, YK97] introduced parallel extensions to the approach presented in [KS95]. [MLI00] presented an improvement by considering a balanced tree (based on red-black trees). However, this method is still main-memory resident. Finally, [YW01] and [ZMT+01] 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 ([Mat94, BKO+97, AE98]). Most solutions are based on the *range tree* proposed by [Ben80]. A variation of the range tree which is used to solve the d-dimensional dominance-sum query is called the *ECDF-tree* ([Ben80]). 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, [ZTG+01] proposed two versions of the *ECDF-B-tree*. [ZTG+01] also presented the *BA-tree* which combines the benefits of the two ECDF-B-trees.

Data Warehousing. The time hierarchy we used is similar to the concept hierarchy in the data warehousing study. [PTK+02] 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. [PTK+02] 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. [GLY98] 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.

8 Conclusions

Temporal aggregates 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. Analytical and experimental results showed the efficiency of the proposed structures. Moreover, we discussed how our solutions can be extended to solve the more general range temporal and spatio-temporal aggregation problems under the fixed time window model.

References

[AE98] 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.

[Ben80] J. L. Bentley, "Multidimensional Divide-and-Conquer", *Communications of the ACM* 23(4), pp. 214-229, 1980.

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

[BKO+97] 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.

[DH00] P. Domingos and G. Hulten, "Mining high-speed data streams", *Proc. of SIGKDD*, pp. 71-80, 2000.

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

[GKS01] J. Gehrke, F. Korn and D. Srivastava, "On Computing Correlated Aggregates over Continual Data Streams", *Proc. of SIGMOD*, 2001.

[GKS01b] S. Guha, N. Koudas and K. Shim, "Data-Streams and Histograms", *Proc. of STOC*, pp. 471-475, 2001.

[GLY98] H. Garcia-Molina, W. Labio and J. Yang, "Expiring Data in a Warehouse", *Proc. of VLDB*, pp. 500-511, 1998.

[GMM+00] S. Guha, N. Mishra, R. Motwani and L. O'Callaghan, "Clustering Data Streams", *Proc. of FOCS*, pp. 359-366, 2000.

[HHW97] J. Hellerstein, P. Haas and H. Wang, "Online Aggregation", *Proc. of SIGMOD*, pp. 171-182, 1997.

[HRR98] M. R. Henziger, P. Raghavan and S. Rajagopalan, "Computing on Data Streams", *TechReport 1998-011, DEC*, May 1998.

[JMS95] H. V. Jagadish, I. S. Mumick and A. Silberschatz, "View maintenance issues for the chronicle data model", *Proc. of PODS*, pp. 113-124, 1995.

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

[LM01] I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *Proc. of SIGMOD*, 2001.

[Mat94] J. Matoušek, "Geometric Range Searching", *Computing Surveys* 26(4), pp. 422-461, 1994.

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

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

[PTK+02] D. Papadias, Y. Tao, P. Kalnis and J. Zhang, "Indexing Spatio-Temporal Data Warehouses", *Proc. of ICDE*, 2002.

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

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

[YW01] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", *Proc. of ICDE*, 2001.

[ZMT+01] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", *Proc. of PODS*, 2001.

[ZTG+01] D. Zhang, V. J. Tsotras, D. Gunopulos and A. Markowetz, "Efficient Aggregation over Objects with Extent", *TechReport UCR_CS_01_01, CS Dept., UC Riverside*, 2001. http://www.cs.ucr.edu/~donghui/publications/boxaggr_full.ps