Temporal Aggregation Using a Multidimensional Index

Joon-Ho Woo, Korea Advanced Institute of Science and Technology (KAIST), Korea

Min-Jae Lee, Korea Advanced Institute of Science and Technology (KAIST), Korea

Kyu-Young Whang, Korea Advanced Institute of Science and Technology (KAIST), Korea

Woong-Kee Loh, Korea Advanced Institute of Science and Technology (KAIST), Korea

Byung Suk Lee, University of Vermont, USA

ABSTRACT

We present a new method for computing temporal aggregation that uses a multidimensional index. The novelty of our method lies in mapping the start time and end time of a temporal tuple to a data point in a two-dimensional space, which is stored in a two-dimensional index, and in calculating the temporal aggregates through a temporal join between the data in the index and the base intervals (defined as the intervals delimited by the start times or end times of the tuples). To enhance the performance, this method calculates the aggregates by incrementally modifying the aggregates from that of the previous base interval without re-reading all tuples for the current base interval. We have compared our method with the SB-tree, which is the state-of-the-art method for temporal aggregation. The results show that our method is an order of magnitude more efficient than the SB-tree method in an environment with frequent updates, while comparable in a read-only environment as the number of aggregates calculated in a query increases.

Keywords: multidimensional index; temporal aggregation

INTRODUCTION

Temporal aggregation is an operation for finding the aggregate value of an attribute over a certain period of time. Specifically, it finds the time intervals in which the aggregate value does not change, namely the constant intervals (Kline & Snodgrass, 1995), and performs the aggregation at each constant interval. Typically, there are two kinds of aggregate functions: cumulative (e.g., COUNT, SUM, AVG) and selective (e.g., MIN, MAX).

There have been several temporal aggregation methods proposed to date. The

early ones included the linked list method by Tuma (1992) and the aggregation tree method by Kline and Snodgrass (1995) and its variants (Gendrano, Huang, Rodrigue, Moon, & Snodgrass, 1999; Kim, Kang, & Kim, 1999; Moon, Lopez, & Immanuel, 2000; Ye & Keane, 1997). Although these methods do facilitate computing temporal aggregates, they require the data structures to reside in main memory. The data structures, however, are typically much larger than the available main memory because a temporal database retains all tuples from the past. Moreover, they require one data structure for each aggregate function.

Recently, Yang and Widom (2003) proposed a method using the SB-tree. This method is similar to the aggregation tree method, but uses a *disk-resident* data structure. (It, however, still requires one data structure for each aggregate function.) In this method, every time a new tuple is inserted or an existing tuple is updated or deleted, the temporal aggregates are updated *immediately* using the SB-tree. Then, queries are executed quickly by simply reading the precomputed aggregate values. However, the overhead of immediate updates is nontrivial, particularly because the update is done for each aggregate function through a separate SB-tree. Thus, this method is not suitable in an environment with frequent insertions, deletions, or updates (collectively called updates from now on) of tuples and relatively less frequent aggregation queries.

Many temporal database applications, however, are update intensive. Examples include financial applications handling stock market data and reservation systems for airlines, hotels, trains, and so forth. For these applications, there are very frequent updates and only a limited number of temporal aggregation queries. Thus, the ratio between updates and aggregation queries in these cases could be in the order of thousands.

In this paper, we propose a new method that resolves the problems of the existing methods for update-intensive applications, while accomplishing reasonably efficient temporal aggregation. Like the SB-tree method, our method uses a disk-resident data structure that is applicable to a large temporal database. The novelty of the method lies in mapping the start time and the end time of a temporal tuple to a data point in a two-dimensional space and storing the data point through a multidimensional index. (In this regard, we call this method the *multidimensional index (MD-index) method* (Trujillo, Luján-Mora, & Song, 2004).) An update operation incurs only a small overhead of inserting a tuple through the index. For an aggregation query operation, aggregates are computed through a temporal join between the data in the index and the *base intervals* (to be defined in the Temporal Aggregation Using a Multidimensional Index section), constituting constant intervals. For efficiency's sake, this calculation is done by incrementally modifying the aggregate from that of the previous base interval without re-reading all tuples for the current base interval.

Compared with the SB-tree method, our method (1) incurs little overhead when updating tuples and, thus, is more efficient for update operations; and (2) uses only one index structure for *all* aggregate functions and, consequently, achieves increasingly comparable aggregation query performance (through incremental calculations) as the number of aggregates in the query increases.

In this paper, "time" may be interpreted as any of the valid time, transaction time, and user-defined time, but we do not con-

sider more than one of them at the same time. In other words, we do not consider bitemporal databases (Jensen et al., 1998). Moreover, the semantics of time is point-based atelic[1]. This means that a tuple value is true at any point within a given time interval, not as a result of achieving a certain goal during the interval.

Experimental results show that our method is an order of magnitude more efficient than the SB-tree method for update operations, and the gap becomes wider as more aggregate functions (e.g., COUNT, SUM, AVG, MIN, MAX) are supported by the system. The results also show that multiaggregation query performance approaches that of the SB-tree method as the number of the aggregate functions increases, becoming comparable to that of the SB-tree, when the query specifies all five aggregate functions (i.e., COUNT, SUM, AVG, MIN, MAX).

Following the Introduction, we provides some background information, followed by a discussion of related work. The Temporal Aggregation Using a Multidimensional Index section elaborates on the proposed MD-index method. The Performance Evaluation section compares the performance between the MD-index method and the SB-tree method, then we provide a conclusion for the paper.

BACKGROUND

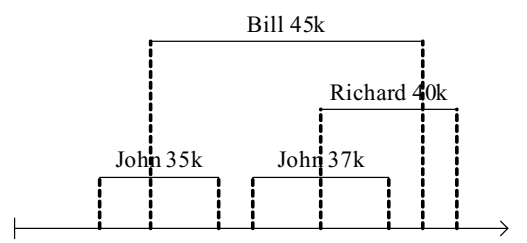
Temporal Aggregation

Each tuple in a temporal relation has an associated time interval (Kline & Snod-

Figure 1. An example of a temporal relation

Name	*Salary*	*Time Interval*
Richard	40k	[18, 25)
John	37k	[14, 21)
John	35k	[5, 12)
Bill	45k	[8, 23)

a) Table E_SALARY.



b) Time intervals of tuples in E_SALARY.

grass, 1995). Figure 1a shows an example of a temporal relation E_SALARY, which stores the salary history of employees. The attribute *Time Interval* defines [*Start time*, *End time*) of tuples as shown Figure 1b[2]. Figure 2 shows the COUNT and MIN of *Salary* changing over time. Note that each aggregate function generates different constant intervals.

MBR-MLGF

We use the multilevel grid file (MLGF) (Whang & Krishnamurthy, 1991) as the multidimensional index[3]. It is a multilevel extension of a grid file and is similar to the K-D-B-tree (Robinson, 1981)—a disk version of the K-D tree—but using hashing. Specifically, it is a dynamic hash file supporting multikey accesses to data through a multilevel directory tree structure.

An MLGF is made of a balanced tree consisting of a multilevel directory and data pages. Each directory level reflects the space partitioning, and each directory entry consists of a region vector and a pointer to either a data page or a lower-level directory page. A region vector in an *n*-dimensional MLGF consists of *n*-hash values that uniquely identify the region, including its position, shape, and size. The *i*-th hash value is the common prefix of the hash values for the *i*-th attribute of all records in the region. A region for a higher-level directory entry contains all regions in the subtree, rooted by the page and pointed by the entry.

Figure 3 illustrates a two-level MLGF with two keys, k_1 and k_2. Figure 3a shows the directory structure, where the two levels are denoted by D_1 and D_2. Figure 3b and Figure 3c show the regions represented by D_1 and D_2, respectively. Here, each region corresponds to a disk page. There are 11 entries in D_1, one for each of the 11 regions A through K, and four entries in D_2, one for each of the D_1 regions a through d. For example, the region vector < 01, 1 > of the directory entry in D_2 represents the region

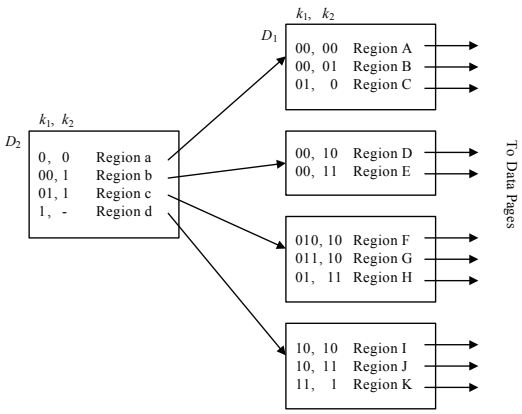
Figure 2. Temporal aggregates COUNT and MIN on E_SALARY.Salary

Constant Interval	*COUNT*	*Constant Interval*	*MIN*
[5, 8)	1	[5, 12)	35k
[8, 12)	2	[12, 14)	45k
[12, 14)	1	[14, 21)	37k
[14, 18)	2	[21, 25)	40k
[18, 21)	3		
[21, 23)	2		
[23, 25)	1		
a) COUNT.		b) MIN.	

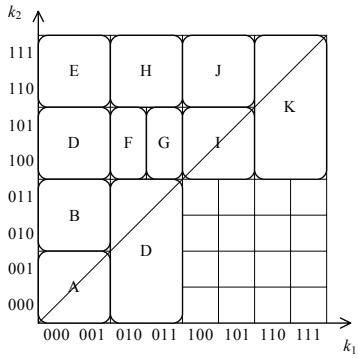
c, in which the two key values are prefixed with 01 and 1, respectively. This region is in turn split into the regions F, G, and H by D_1, each identified with the directory entry with the region vector < 010, **10 >, < 011, 10** >, or < 01, **11 >. In the MBR-MLGF, each** directory entry maintains information about the minimum bounding regions of objects (without additional storage overhead). For example, Figure 4 shows the region R_1 in a rectangle and the objects in it as points. The vertical line at *min-*t_s and horizontal line at *max-*t_e reduces R_1 to its MBR.

We particularly use the MBR-MLGF (minimum-bounding-region MLGF) (Song, Shang, Lee, Lee, & Kim, 1999), which is an extension of the MLGF targeted toward efficient spatiotemporal data accesses. In the MBR-MLGF, each directory entry maintains information about the minimum bounding regions of objects (without additional storage overhead). For example, Figure 4 shows the region R_1 in a rectangle and the objects in it as points. The vertical line at *min-*t_s and the horizontal line at *max-*t_e reduces R_1 to its MBR.

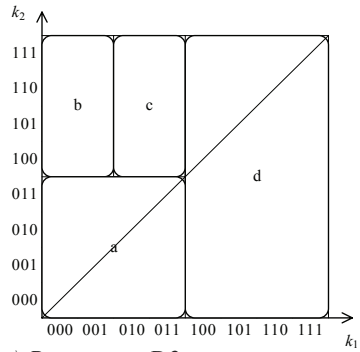
Figure 3. An example of a MLGF



a) A two-level directory structure



b) Regions in D1



c) Regions in D2

RELATED WORK

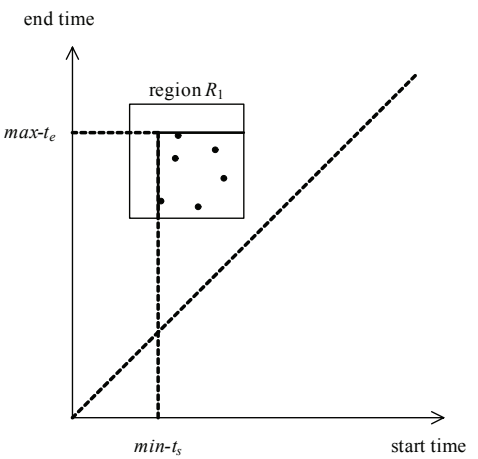
The first proposal for computing temporal aggregates has been made by Tuma (1992). It was based on an extension to the nontemporal aggregate computation algorithm by Epstein (1979). The approach consists of two steps, each requiring one scan of the base table. The first step partitions the time line into constant intervals. The second step considers each tuple *t* in the base table in turn, updating the aggregate values for all resultant tuples, covered by the tuple *t*'s valid interval. This method takes $O(mn)$ time to compute temporal aggregates, where *n* is the size of the base table and *m* is the number of result tuples. This method builds a linked list which resides in main memory to represent constant intervals that are generated in the first step and used in the second step.

Kline and Snodgras (1995) proposed the aggregation tree based on the binary segment-tree (Preparata & Shamos, 1985). This segment-tree feature allows efficient processing of tuples with long intervals. The tree structure, however, is unbalanced. In the worst case, it takes $O(n^2)$ to compute a temporal aggregate from a table with *n* tuples. There has been balanced aggregation trees (Kim et al., 1999; Moon et al., 2000) proposed so that the worst-case run time is $O(n \log m)$, where *m* is the number of the constant interval. There have been parallel versions of the aggregation-trees (Ye & Keane, 1997; Gendrano et al., 1999), but they all inherit the same limitations of the original (i.e., nonparallel) versions.

One major drawback of the methods described so far is that they use main-memory resident data structures. It limits the applicability of the methods to a temporal database that is often too large to fit in main memory.

Yang and Widom (2003) proposed the SB-tree based on the segment-tree (Preparata & Shamos, 1985) and the B-tree.

Figure 4. Minimum bounding region in an MBR-MLGF



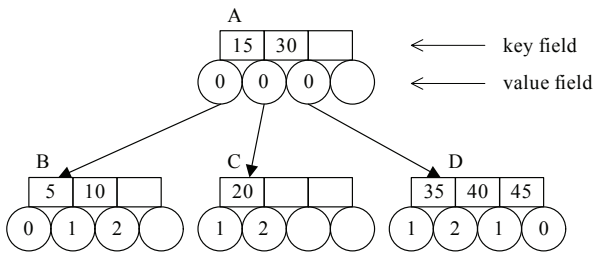
Although similar to the aggregation tree in terms of storing a time interval and a value in each node, the SB-tree is different for storing multiple time intervals in each node. More importantly, the SB-tree is a disk-resident data structure.

Figure 5 shows an example SB-tree for COUNT. Each node of the SB-tree consists of key-value pairs. Each key is the start time of a constant interval. Note that the end time of one interval is the same as the start time of the next interval. Each value is the value for the constant interval. For example, in Figure 5a, the node B contains three constant intervals [0, 5), [5, 10) and [10, 15), and the values of COUNT are 0 in [0, 5), 1 in [5, 10) and 2 in [10, 15).

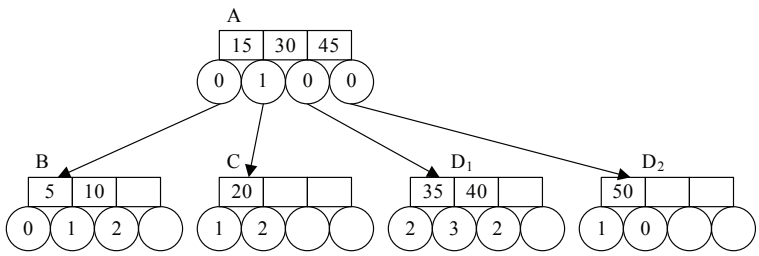
A key insertion of SB-tree is handled as in the B-tree, and a value-update is done to a node at the highest possible level. For example, let us assume we insert a tuple with the time interval [10, 50) and, as a result, two keys 10 and 50 are inserted into the SB-tree shown in Figure 5a. Since the key 10 already exists, only 50 is inserted. This insertion causes the node D to overflow. Then, (as in the B-tree) the node D is split into the nodes D_1 and D_2 as shown in Figure 5b, and the keys are redistributed to the two new nodes while the center value 45 is moved to the parent. Besides, 1 is added to the values in all nodes whose keys lie within the interval [10, 50). Note in Figure 5b that 1 is added to the second value in the parent node A instead of the first and the second values in the node C, since the entire interval [15, 30) is included in the interval [10, 50). Figure 6 shows how the SB-tree is constructed as the tuples in Figure 1a are inserted.

All the methods described in this section share a common disadvantage and advantage. The disadvantage is that they require one data structure for each aggregate

Figure 5. An example of the SB-tree for COUNT



a) Before inserting [10, 50)



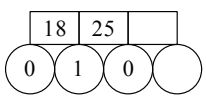
b) After inserting [10, 50)

function. The advantage is that, when an aggregation query is issued, they have only to retrieve the up-to-date aggregate values in the data structures immediately (with an exception of the linked list method (Tuma, 1992)). As mentioned in the Introduction, our MD-index method has the opposite disadvantage and advantage. That is, it requires only one data structure for all aggregate functions, and computes the aggregate values at query time.

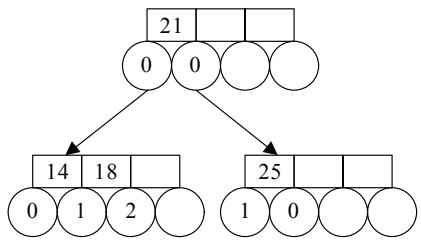
Temporal Aggregation Using a Multi-dimensional Index

As mentioned in the Introduction, we represent temporal tuples as points in a two-dimensional (2-D) space defined by the start time and end time of the tuples. This mapping enables the proposed MD-index method. Based on this concept, we define the temporal join window and present the aggregation algorithms.

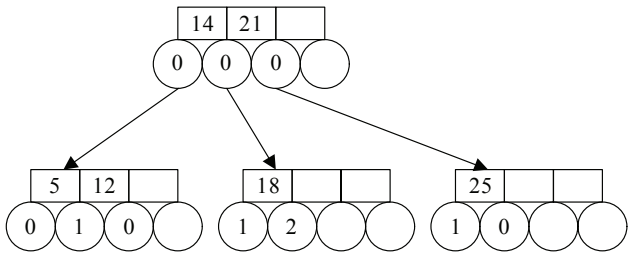
Figure 6. An example of the SB-tree construction



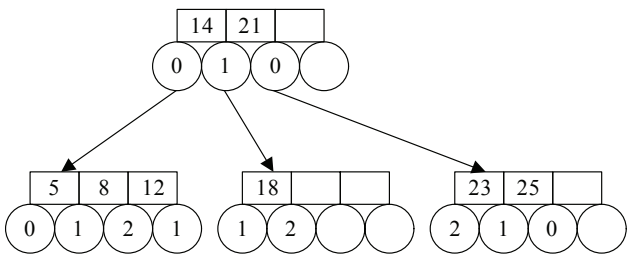
a) After inserting [18, 25)



b) After inserting [14, 21)



c) After inserting [5, 12)



d) After inserting [8, 23)

Temporal Join Windows

We first define the base interval as follows.

Definition 1 (base intervals): Given temporal tuples, their base intervals are the time intervals delimited by the start times or end times of all tuples. □

For a given aggregate function, if we merge all adjacent base intervals with the same aggregate values, then the merged intervals compose one constant interval (Kline & Snodgrass, 1995) for the aggregate function. Base intervals are maintained by storing the start time and end time of each tuple in a separate B+-tree.

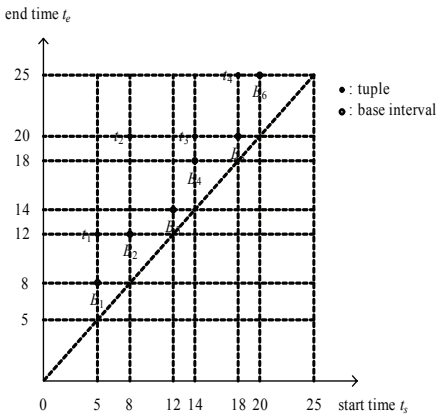
As the time interval of a tuple can be mapped to a 2-D point, so can the base interval be. Tuples thus mapped to 2-D points can be stored and retrieved through a 2-D index. Figure 7a shows the time intervals of four tuples and six base intervals of COUNT, and Figure 7b shows the 2-D points mapped from these intervals.

Definition 2 (temporal join window of a base interval): In a 2-D space representing all possible temporal tuples, we define the temporal join window (TJW) of a base interval B_i (TJW(B_i)) as the 2-D region containing all tuples whose time intervals overlap B_i. That is, given $B_i \equiv [s_i, e_i)$,

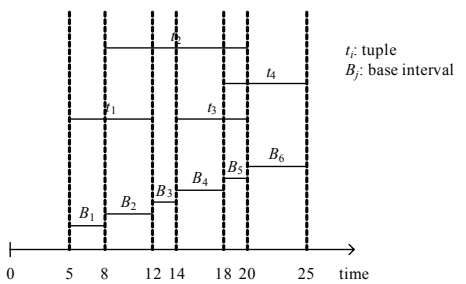
$$\mathrm{TJW}(B_i) = \{< t_s, t_e > \mid t_s \le s_i \text{ and } t_e \ge e_i\}$$

For example, in Figure 7b, TJW(B_4) contains the tuples t_2 and t_3, and the tuple t_2 belongs to TJW(B_2), TJW(B_3), TJW(B_4), and TJW(B_5). Note that the tuples' points are located only at the grid points formed by the TJWs because, by definition, there cannot be the start time or end time of any tuple within a base interval.

Figure 7. Tuples and base intervals



a) Intervals in the 1-D time



b) Points in the 2-D space

Temporal Aggregation Algorithms

Temporal aggregate for each base interval, B_i, is obtained by aggregating tuples overlapping B_i, that is, tuples in TJW(B_i). As we can see from Figure 8, aggregating tuples in the order of B_{i-1}, B_i, B_{i+1} allows reusing tuples from the previous TJW. We now present the algorithms for computing aggregates in the order of base intervals. (It may well be done in the reverse order.)

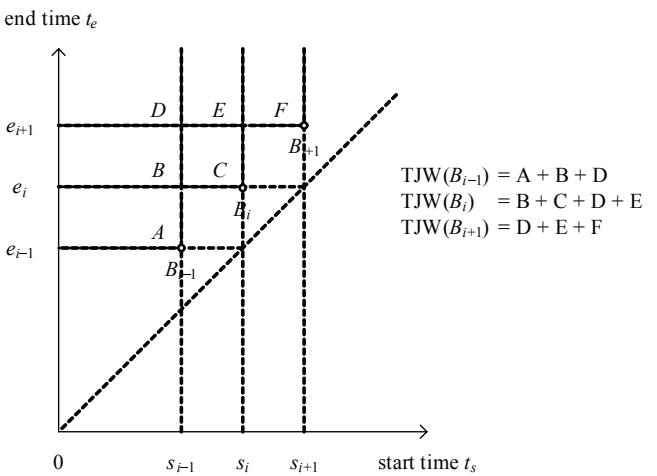
The algorithm differs between cumulative aggregation and selective aggregation. Let us first consider the cumulative one using COUNT as an example. (SUM is obtained in the same way as COUNT, and AVG is obtained as SUM divided by COUNT.) In Figure 8, COUNT of tuples in TJW(B_i) is obtained from the COUNT of tuples in TJW(B_{i-1}) by adding the COUNT of tuples in the region C+E and subtracting the COUNT of tuples in A.

Because tuples' points are located only at the grid points formed by the TJWs, all tuples in the region A have the end time e_{i-1} and all tuples in the region C have the start time s_i. Let $N_s(s_i)$ be the number of tuples with the start time s_i and $N_e(e_{i-1})$ be the number of tuples with the end time e_{i-1}. Then, COUNT(B_i), the value of COUNT in the base interval B_i, is obtained as

$$\text{COUNT}(B_i) = \text{COUNT}(B_i\text{-}1) - N_e(e_i\text{-}1) = N_S(S_i) \quad (1)$$

Now, let us consider the selective aggregation with MIN as an example. MAX is symmetric to MIN and, therefore, is obtained in the same way as MIN. When calculating MIN in B_i after B_{i-1}, there are two cases depending on the MIN of the tuples in A. In case the MIN in B_{i-1} is different from the MIN in A, the tuple with the minimum value must be in B+D and, therefore, the MIN in B_i is the smaller of MIN in B+D (= MIN in B_{i-1}) and that in C+E. In case the MIN in B_{i-1} and the MIN in A are the

Figure 8. TJWs of three consecutive base intervals



same, the tuple with the minimum value must be in A and, therefore, the MIN in B_i is the smaller between the MIN of tuples in A and that in C+E.

Let $M_s(s_i)$ be the minimum value of the tuples with the start time s_i and $M_e(e_{i-1})$ be the minimum value of the tuples with the end time e_{i-1}. Then, MIN(B_i), the value of MIN in the base interval B_i, is obtained as

$$\text{MIN}(B_i)= \begin{cases} \text{LESSER}(\text{MIN}(B_{i-1}), M_s(s_i)), & \text{if MIN}(B_{i-1}) \neq M_e(e_{i-1}) \\ \text{LESSER}(M_e(e_{i-1}), M_s(s_i)), & \text{otherwise} \end{cases} \quad (2)$$

where the function LESSER returns the smaller of the two arguments.

Figure 9 outlines the algorithms for calculating COUNT and MIN based on Equations (1) and (2) given a series of base intervals. The aggregate for the first base interval is computed in line 1 of each algorithm. Then, the aggregates for the rest of the base intervals are calculated incrementally in lines 2-15 and 2-17, respectively. It outputs one aggregate value per constant interval, resulting from merging adjacent base intervals in lines 7-11 lines 9-13, respectively.

Figure 9. COUNT and MIN aggregation algorithms

```
Algorithm Aggregate_COUNT
  Input: A series of base intervals B_i = [s_i, e_i), i=1,2,…,n
  Ouput: COUNT aggregate values for each constant interval C
  begin
   1:  prev_count := the number of tuples with the start time ≤ s_1 and the end time ≥ e_1;
   2:  s := B_1's start time s_1;
   3:  N_e := the number of tuples with the end time = e_1;
   4:  for i := 2 to n {
   5:     N_s := the number of tuples with the start time = s_i;
   6:     count := prev_count – N_e + N_s;
       /* if (count == prev_count) then merge the base intervals to a constant interval */
       /* else output the result: */
   7:     if count != prev_count {
   8:        output <[s, e_{i-1}), prev_count>;
   9:        s := B_i's start time s_i;
  10:     }
  11:     e := B_i's end time e_i;
  12:     N_e := the number of tuples with the end time = e_i;
  13:     prev_count := count;
  14:  }
  15:  output <[s, e_n), count>;
  end;
```

a) Aggregate_COUNT

PERFORMANCE EVALUATION

We have conducted experiments to compare the performance with that of the SB-tree method. We describe the experimental setup and present the results in this section.

Experiment Setup

We use two synthetic datasets (DS1, DS2) generated in a manner similar to the data used by Kline and Snodgrass (1995) and Moon et al. (2000). There are four temporal relations. Each tuple has four attributes: name (4 bytes), salary (4 bytes), start time (4 bytes), and end time (4 bytes). The relation sizes are 1, 4, 16, and 64 Mbytes, each of which contains 65,536, 262,144, 1,048,576, and 4,194,304 tuples.

The tuples in DS1 are uniformly distributed with respect to time. Their start time is selected randomly between 1 and the following time range (inclusive): 1 million for the 1 Mbyte relation, 4 million for the 4 Mbyte relation, and so forth. (Note that we use an integer for the time stamp.) The end time is selected randomly between the start time + 1 and the start time + 30% of the time range. The tuples in DS2 are normally distributed, thus skewed, with respect to time. The mean and the standard deviation are 1/4 and 1/8 of the time range for the start time, and 3/4 and 1/8 of the time range for the end time.

After inserting tuples in a 1 Mbyte relation, for example, the size of the MD-index is 3.2 Mbytes, and the size of SB-tree

Figure 9. continued

```
Algorithm Aggregate_MIN
  Input: A series of base intervals B_i = [s_i, e_i), i=1,2,…,n
  Ouput: MIN aggregate values for B_i, i=1,2,…,n
  begin
   1:  prev_min := the minimum value of the tuples with the start time ≤ s_1 and the end time ≥ e_1;
   2:  s := B_1's start time s_1;
   3:  M_e := the minimum value of the tuples with the end time e_1;
   4:  for i := 2 to n {
   5:     M_s := the minimum value of the tuples with the start time s_i;
   6:     if (prev_min != M_e)
   7:        then min := lesser(prev_min, M_s);
   8:     else min := lesser(M_e, M_s);
         /* if (min == prev_min) then merge the base intervals to a constant interval */
         /* else output the result: */
   9:     if min != prev_min {
  10:        output <[s, e_i-1), prev_min>;
  11:        s := B_i's start time s_i;
  12:     }
  13:     e := B_i's end time e_i;
  14:     M_e := the minimum value of the tuples with the end time e_i;
  15:     prev_min := min;
  16:  }
  17:  output <[s, e_n), min>;
  end;
```

b) Aggregate_MIN.

is 2.4 Mbytes for SUM, 2.3 Mbytes for COUNT, 2.4 Mbytes for AVG, 0.8 Mbytes for MIN, and 0.8 Mbytes for MAX. The size of the SB-tree varies significantly between cumulative and selective aggregate functions. Considering a 16 Mbyte relation as another example, the SUM SB-tree is approximately 11 Mbytes, while the MIN SB-tree is only 12 Kbytes. This is because MIN typically generates a smaller number of constant intervals (see Figure 3).

We measure the update performance by inserting new tuples amounting to 0.1% of the existing tuples into the datasets DS1 and DS2. The ratio of 0.1% is sufficient for our purpose because, evidently, the performance gap between the MD-index method and the SB-tree method would increase as more tuples are inserted. The maintenance cost of base intervals in the B+-tree is also included in the update cost. The tuples have been inserted in a batch but without any optimization taking advantage of the batch processing, so the result would not differ much from that of inserting tuples in increments.

The performance metrics are the elapsed time and the number of disk page accesses. The elapsed time is the total execution time measured in a single-user environment. We consider only the five standard aggregate functions (i.e., SUM, COUNT, AVG, MIN, MAX). In order to avoid noise, we execute each function more than three times and calculate an average.

The system is configured in Linux server with 1.0 Gbyte RAM and ATA-4 IDE hard disk drive, and it uses direct I/O to eliminate the unpredictable effect of operating system buffering. The page size is 4 Kbytes for both disk pages and buffer pages.

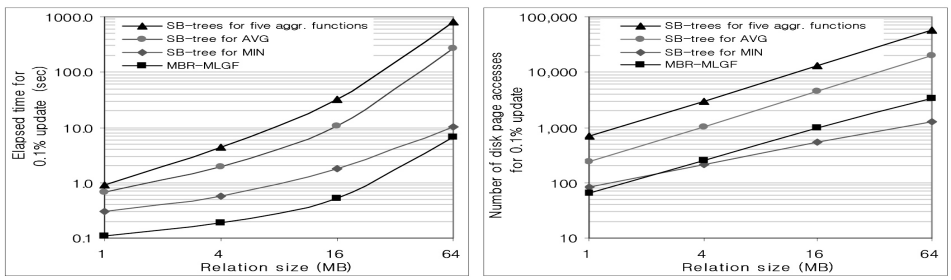
Experiment Results

Experimental performance results of update operations and aggregation query operations are presented in this subsection.

Update Performance

Figures 10 and 11 compare the update performances of our MD-index method and the SB-tree method, using the dataset DS1 and the dataset DS2, respectively. Each figure shows two cases for the SB-tree method: the five aggregate functions together and

Figure 10. Update performance with respect to relation size using DS1



each aggregate function separately. In the latter case, we show the results for only AVG and MIN because the results for SUM and COUNT are very close to that for AVG, and the result for MAX is very close to that for MIN.

As mentioned in the Introduction, the SB-tree method requires one SB-tree for each aggregate function, whereas our method uses one MD-index for all. Therefore, each update operation incurs updating five SB-trees in the SB-tree method, whereas it incurs updating one MBR-MLGF in the MD-index method. Figures 10 and 11 show that the MD-index method performs far better (by one to two orders of magnitude) than the SB-tree method when all the five aggregate functions are considered.

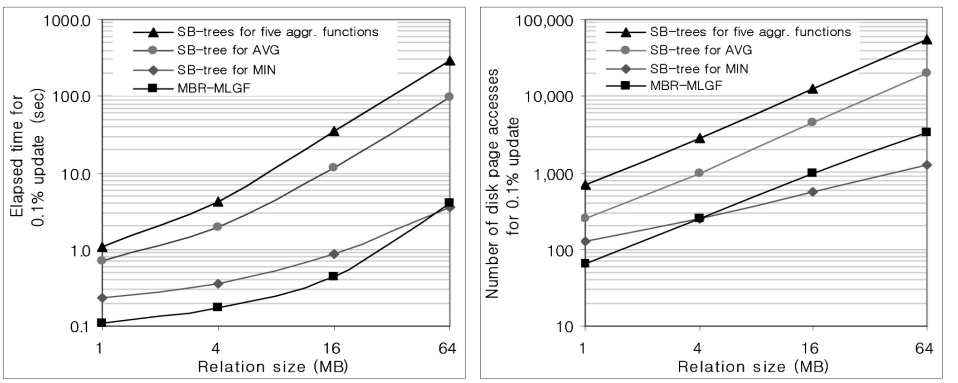
The two figures also show that the update performance of the SB-tree method is much better for MIN than AVG. While the AVG performance is much poorer than that of the MD-index method by an order of magnitude, the MIN performance is comparable. Moreover, the update costs of MIN SB-tree increases slower than those of the other three. The reason is the size of the MIN SB-tree, which is much smaller than the AVG SB-tree or the MD-index.

Aggregation Query Performance

Figures 12 and 13 compare the aggregation query performance of our MD-index method and the SB-tree method using the dataset DS1 and dataset DS2, respectively. Here, each figure also shows two cases for the SB-tree method: the five aggregate functions together and each aggregate function separately.

The two figures show that the aggregation query performance of the MD-index method is worse than that of the SB-tree method for a single aggregate function. This is as expected. We also see that the gap is larger for the MIN aggregate function due to the smaller size of the MIN SB-tree. Oftentimes, multiple aggregate functions appear in the same aggregation query for periodic statistics reports, for example. This multiaggregation case brings a performance advantage to the MD-index method because it calculates all aggregates while accessing the index tree only once, regardless of the number of the aggregate functions. Indeed,

Figure 11. Update performance with respect to relation size using DS2



the two figures show that the performance of the MD-index method is comparable to (and, in some cases, better than) that of the SB-tree method when all the five aggregate functions are considered together.

CONCLUSION

We have presented a new temporal aggregation method called the MD-index method. It stores temporal tuples as 2-D points through a 2-D index; and it computes the aggregates by identifying the TJW of each base interval and joining the tuples in the window with the interval. The aggregates for base intervals are calculated by incrementally modifying the aggregates from the previous base intervals without rereading all tuples in the TJW of the current base interval. Adjacent base intervals with the same aggregate value are subsequently merged into a constant interval. We have compared our method with the SB-tree method. The results show that our method is at least an order of magnitude faster than the SB-tree method for updates, while increasingly comparable for multiaggrega-

Figure 12. Temporal aggregation query performance with respect to relation size using DS1

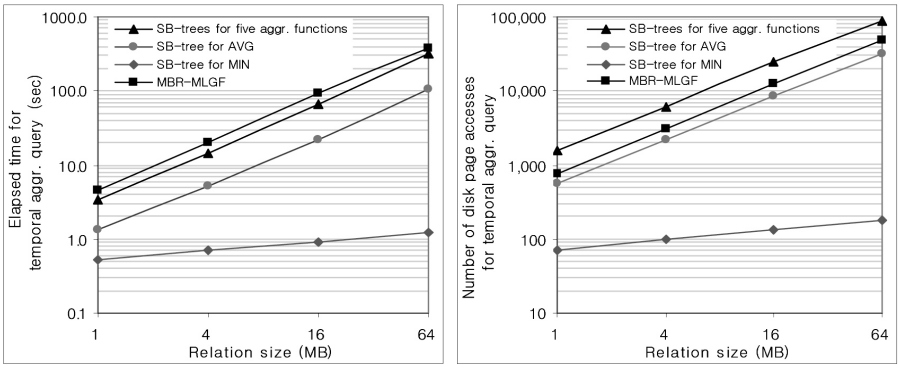
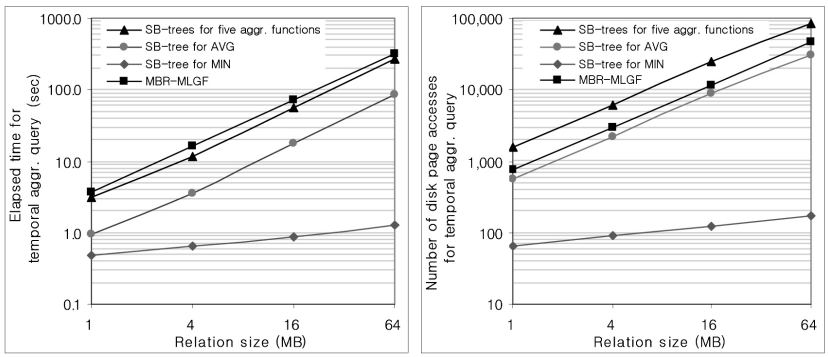


Figure 13. Temporal aggregation query performance with respect to relation size using DS2



tion queries as the number of aggregate functions in the query increases. These results indicate that the MD-index method is preferable in an environment with frequent updates or multiaggregation queries.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their invaluable comments. This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc) and the Brain Korea 21 Project. For the author Byung Suk Lee, the work was partially supported by the U.S. Department of Energy through Grant No. DE-FG02-ER45962 and the U.S. National Science Foundation through Grant No. IIS-0415023.

REFERENCES

Epstein, R. (1979). *Techniques for processing of aggregates in relational database systems* (Tech. Rep. UCB/ERL M7918). Berkeley, CA: University of California.

Gaede, V., & Gunther, O. (1998). Multidimensional access methods. *ACM Computing Surveys, 30*(2), 170-231.

Gendrano, J. A. G., Huang, B. C., Rodrigue, J. M., Moon, B., & Snodgrass, R. T. (1999). Parallel algorithms for computing temporal aggregates. In *Proceedings of the IEEE International Conference on Data Engineering, Sydney, Australia,* 418-427.

Jensen, C. S., et al. (1998). The consensus glossary of temporal database concepts (February 1998 version). Temporal databases: research and practice. *Lecture Notes in Computer Science (LNCS), 1399,* 373-374.

Kim, J., Kang, S., & Kim, M. (1999). Effective remporal aggregation using point-based trees. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA), Florence, Italy,* 1018-1030.

Kline, N., & Snodgrass, R. T. (1995). Computing remporal aggregates. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE), Taipei, Taiwan,* 222-231.

Moon, B., Lopez, I. F. V., & Immanuel, V. (2000). Scalable algorithms for large temporal aggregation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE), San Diego, USA,* 145-154.

Preparata, F. P., & Shamos, M. I. (1985). *Computational geometry: An introduction*. Berlin/Heidelberg, Germany: Springer-Verlag.

Robinson, J. T. (1981). The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the International Conference on Management of Data, ACM SIGMOD, New York, USA,* 10-18.

Song, J., Whang, K., Lee, Y., Lee, M., & Kim, S. (1999). Spatial join processing using corner transformation. *IEEE Transactions on Knowledge and Data Engineering, 11*(4), 688-698.

Terenziani, P., & Snodgrass, R. T. (2004). Reconciling point-based and Interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Transactions on Knowledge and Data Engineering, 16*(5), 540-551.

Trujillo, J., Luján-Mora, S., & Song, I.-Y. (2004) .Applying UML and XML for designing and interchanging information for data warehouses and OLAP

applications. *Journal of Database Management, 15*(1), 41-72.

Tuma, P. A. (1992). *Implementing historical aggregates in TempIS*, Unpublished master's thesis, Wayne State University, .

Whang, K., Kim, S., & Wiederhold, G. (1994). Dynamic maintenance of data distribution for selectivity estimation. *The VLDB Journal, 3*(1), 29-51.

Whang, K., & Krishnamurthy, R. (1991). The multilevel grid file a dynamic hierarchical multi-dimesional file structure. In *Proceedings of the 2nd International Conference on Database Systems for Advanced Applications (DASFAA), Tokyo, Japan,* 449-459.

Yang, J., & Widom, J. (2003). Incremental computation and maintenance of temporal aggregates. *The VLDB Journal, 12*(3), 262-283.

Ye, X., & Keane, J. A. (1997). Processing temporal aggregates in parallel. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, Orlando, USA,* 1373-1378.

Zhang, D., Markowetz, A., Tsotras, V., Gunopulos, D., & Seeger, B. (2001). Efficient computation of temporal aggregate with range predicates. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Santa Barbara, USA.*

ENDNOTES

1 An atelic fact is characterized by being true at any point during a certain time interval, whereas a telic fact is characterized by being true as a result of completing a goal during an interval (Terenziani & Snodgrass, 2004).

2 We represent the time as an integer for convenience without loss of generality.

3 Nevertheless, any multidimensional point access method (Gaede & Gunther, 1998) can be used for this purpose.

Joon-Ho Woo received his BS degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1993 and earned his MS and PhD degrees in Computer Science from KAIST in 1995 and 2004, respectively. He worked at Center for Advanced Information System (CAIS), KAIST and participated in developing the integrated campus information system. He is currently with Samsung SDS, Co., Ltd. His research interest includes spatio-temporal databases, geographic information system, and information retrieval.

Byung Suk Lee received his BS degree from Seoul National University in 1980, M.S. from Korea Advanced Institute of Science and Technology in 1982, and PhD from Stanford University in 1991. His research areas include database systems, data modeling, and information retrieval. He has held various positions in industry and academia, including

Gold Star Electric, Bell Communications Research, Datacom Global Communications, University of St. Thomas, and currently University of Vermont. He was also a visiting professor at Dartmouth College and a guest professor at Lawrence Livermore National Laboratory. He served on international conferences as a program committee member, publicity chair, and special session organizer, and served on US federal funding programs, including NSF proposal review panel and DOE EPSCoR session panel.

Min-Jae Lee received his BS degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1995, and earned his MS and PhD degrees in Computer Science from KAIST in 1997 and 2004, respectively. Until November 2004, he was a post-doctoral fellow at Advanced Information Technology Information Center, KAIST. In December 2005, He joined Neowiz, Co., Ltd., in Korea as a researcher. His research interest includes spatial databases, access methods, information retrieval, query processors, database systems, and storage systems.

Kyu-Young Whang graduated (Summa Cum Laude) from Seoul National University in 1973 and received his MS degrees from Korea Advanced Institute of Science and Technology (KAIST) in 1975, and Stanford University in 1982. He earned the PhD degree from Stanford University in 1984. From 1983 to 1991, he was a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. In 1990, he joined KAIST, where he currently is a full professor at the Department of Computer Science and the Director of the Advanced Information Technology Research Center (AITrc). His research interests encompass database systems/storage systems, object-oriented databases, multimedia databases, geographic information systems (GIS), data mining/data warehouses, and XML databases. He is an author of over 90 papers in refereed international journals and conference proceedings (and over 140 papers in domestic ones).
He served as an IEEE Distinguished Visitor from 1989 to 1990, received the Best Paper Award from the 6th IEEE International Conference on Data Engineering (ICDE) in 1990, served the ICDE six times as a program co-chair and vice chair from 1989 to 2003, and served program committees of over 90 international conferences including VLDB and ACM SIGMOD. He was the program chair (Asia and Pacific Rim) for COOPIS'98 and the program chair (Asia, Pacific, and Australia) for VLDB 2000. He is a general co-chair of VLDB 2006 and the general chair of PAKDD 2003 and DASFAA 2004. He twice received the External Honor Recognition from IBM. He is an Editor-in-Chief of the VLDB Journal having served the editorial board as a founding member for thirteen years. He was an associate editor of the IEEE Data Engineering Bulletin from 1990 to 1993 and an editor of Distributed and Parallel Databases Journal *from 1991 to 1995. He is on the editorial boards of the* IEEE TKDE, The World-Wide Web Journal*, and* Int'l Journal of GIS. *He was a trustee of the VLDB Endowment from 1998 to 2004. He is the vice chair of the steering committee of the DASFAA Conference and is a steering committee member of the PAKDD Conference. He served the IEEE Computer Society Asia/Pacific Activities Group as the Korean representative from 1993 to 1997.*
He is a senior member of the IEEE, a member of the ACM, and a member of IFIP WG 2.6.

Woong-Kee Loh received his BS degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1991, and earned his MS and PhD degrees in Computer Science from KAIST in 1993 and 2001, respectively. He was a winter student of NHK Science and Technical Research Laboratories (STRL) in Tokyo, Japan in 1995. He was a visiting summer student at Computer Science Department, Stanford University in 1997. Until March 2005, he was a principal researcher at Tmax Data, Co., Ltd. in Bundang, Korea, and worked on a new DBMS architecture that reduces system overheads. Since April 2005, he has been a visiting professor at Computer Science Department, KAIST. His research interest includes data mining/data warehousing, spatio-temporal databases, main memory databases, XML internet databases, and multimedia content-based retrieval.