

Short Paper

A Temporal Aggregation Method for Update-Intensive Applications^{*}

SUNG TAK KANG, YON DOHN CHUNG⁺ AND MYOUNG HO KIM

*Department of Computer Science
Korea Advanced Institute of Science and Technology
Yusung-Gu, Daejeon, 305-701, Korea*

*⁺Department of Computer Science and Engineering
Korea University
Seongbuk-Gu, Seoul, 136-713, Korea*

The temporal aggregation in temporal databases is an extension of the conventional aggregation including the time as a range condition of the aggregation. In this paper, we propose a new tree based structure for the temporal aggregation, called the *CTA-tree*. In the *CTA-tree*, we transform the time interval of a temporal data record into a value, called the *T-value*, using the Corner transformation and column-scan transformation. Then, we index the data records according to their T-values. Through analyses and experiments, we evaluate and compare the performance of the proposed method with the conventional method, and show the efficiency of the proposed method for the update-intensive applications.

Keywords: temporal aggregation, corner transformation, data structures, temporal databases, databases

1. INTRODUCTION

Temporal database (TDB) systems provide built-in supports for efficient storing and querying of time-varying data [1]. The TDB is essential to various applications such as the trend analysis, version management and video data management. It is also popularly used for managing temporal data in data warehouses.

The early research on the TDB mainly the considered conceptual problems such as modeling, query languages, and implementation related problems – the indexing, query processing and storage management [2, 3]. The recent research on the TDB has focused on the temporal operations including the temporal aggregation and their processing techniques [4-9].

The aggregation is an operation that calculates a value or selects a representative value from a (part of) relation. The temporal aggregation extends the conventional aggregation to include the time concept on the domain and the range of aggregation. It is

Received June 2, 2006; revised November 6, 2007 & March 6, 2008; accepted March 13, 2008.

Communicated by Suh-Yin Lee.

^{*} This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R0A-2007-000-10046-0).

⁺ Corresponding author.

essential to the statistical tasks, decision support applications, data warehouses and OLAP. There have been several proposals for the temporal aggregation processing, most of which are tree-based approaches. In the tree-based approach, a special tree structure is built for the records, and the temporal aggregates are computed by traversing the constructed tree structure. Therefore, the efficient tree construction and aggregate computation techniques are the key to the improvement of the performance.

The SB-tree [7] is one of the most popularly used disk-based temporal aggregation method. It stores all possible time intervals that can be made by the given time interval and their aggregate values. It is useful when the time domain is narrow compared with the number of temporal data records. However, this situation is not so usual. In addition, the SB-tree has some drawbacks in update-intensive situations. In the worst case, there can be $2N$ index records in the tree when we insert N temporal data records, and thus, $O(\log N)$ disk updates are needed for one temporal data record insertion.

In this paper, we propose a new tree structure, named the *Corner Transformation-based Aggregation tree (CTA-tree)*, and an aggregate processing method based on the CTA-tree. Through analyses and performance experiments, we show the advantages of the proposed method.

The remainder of the paper is organized as follows. Section 2 describes some preliminary knowledge for our study. Section 3 describes the structure of the CTA-tree and the aggregation processing method. In section 4, we analyze the complexity of the proposed method and evaluate the performance of the proposed method through experiments. Finally, we conclude the paper in section 5.

2. PRELIMINARIES

2.1 Background

The whole time-line is partitioned into a number of intervals based on the time intervals of temporal data, and each partitioned interval may have a different aggregate value. The maximal continuous intervals on which we have the same aggregate values are called constant intervals. A temporal aggregation denotes the operation that makes constant intervals and computes their aggregate values. Therefore, the temporal aggregations can be used to see the changes of aggregate values on time. Fig. 1 shows an example of the temporal aggregation. Fig. 1 (a) is a set of temporal data records, and Fig. 1 (b) is the time diagram of them. Fig. 1 (c) shows the temporal aggregation results for the MAX and COUNT operations based on this data set. Here, the constant intervals for MAX are $[0, 9]$, $[10, 15]$, $[16, 16]$, $[17, 18]$, and so on. Those for COUNT are $[0, 2]$, $[3, 5]$, $[6, 9]$, $[10, 11]$, $[12, 12]$, and so on.

2.2 Related Work

There were many proposals for the temporal aggregation. Some approaches [3-5] were based on the main-memory, and the others [9, 10] were based on the disk. Since the amount of data in the TDB becomes very large, the memory-based approaches are considered inappropriate for real applications. Therefore, in this paper, we focus on the disk-based approaches.

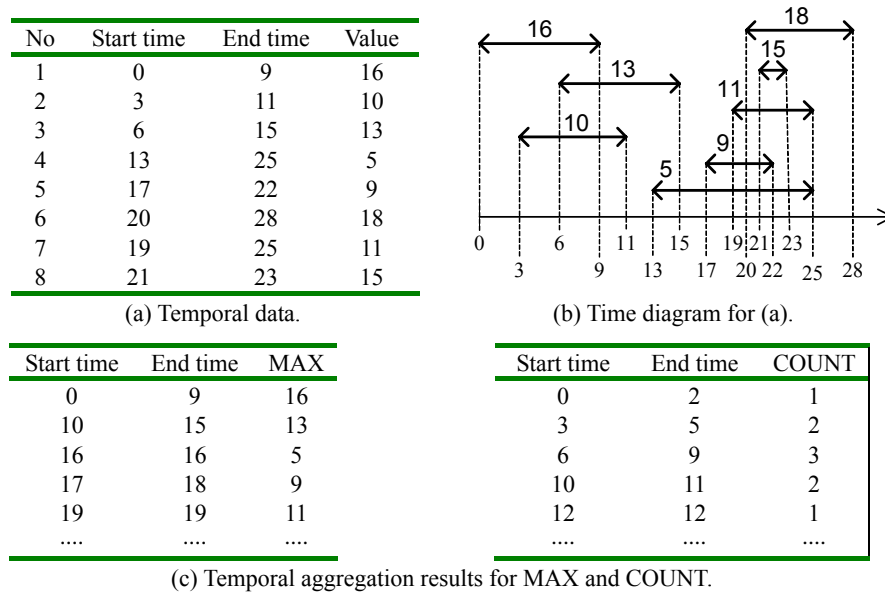


Fig. 1. An example of temporal aggregation.

The *SB-tree* [7] was proposed as a disk-based method for the temporal aggregation and known to be the most efficient [5]. It is based on the *segment-tree* and *B-tree*. The temporal aggregation processing based on the *SB-tree* consists of two steps: the tree construction and aggregate value computation with a depth-first-search on the tree. The time intervals of child nodes partition their parent interval. That is, the time intervals of sibling nodes are disjoint, and the union of them makes their parent interval. The constant intervals and their aggregate values are maintained in the leaf nodes. The internal nodes have time intervals and their aggregate values. Since the aggregate values in the leaf nodes are partial aggregate values, rather than the complete values for the constant intervals, we have to compute the real aggregate values by a depth-first search on the tree *i.e.*, the aggregate values are computed by traversing the nodes on the path from the root to a leaf. Fig. 2 illustrates the construction of the *SB-tree* for the COUNT operation using the data set in Fig. 1 (a).

When a new temporal data record is inserted into the *SB-tree*, new index records are generated by partitioning the time intervals of leaf nodes that intersect the time interval of the new temporal data. (The components in the tree structure are different from the temporal data. For example, a component in the tree contains a time point instead of a time interval in the temporal data. In the paper, we refer to the components in the tree structure as the *index records* for the purpose of distinguishing them from the temporal data records.) There can be at most two new index records in the leaf nodes for the insertion of a single data record. Fig. 2 (a) shows the *SB-tree* after inserting data record 1. There is one index record (9, 1), which means the upper bound of the time interval [0, 9] and its aggregate value. After inserting data record 2, the *SB-tree* becomes as in Fig. 2 (b), where the constant intervals become [0, 2], [3, 9], and [10, 11]. After inserting data record 3, the *SB-tree* becomes as in Fig. 2 (c). The internal node N_1 has one index record

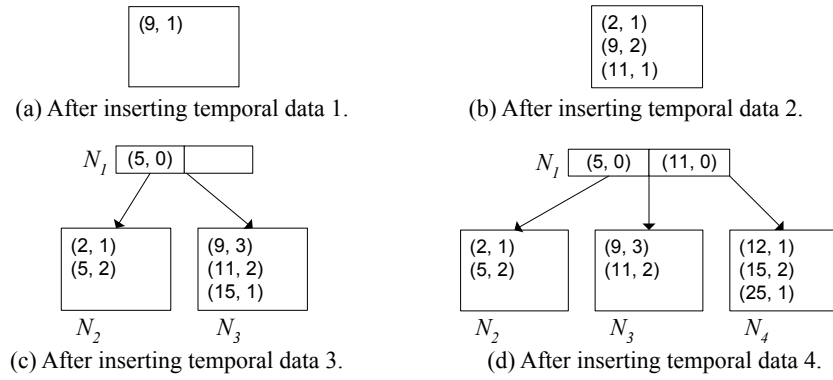


Fig. 2. The construction of the SB-tree for COUNT.

and the index record has the time interval $[0, 5]$ and its aggregate value '0'. The time intervals in the internal nodes are updated when the time interval of a new data record covers them. For example, the aggregate value for the time interval $[0, 5]$ becomes 1 when a new temporal data with a time interval $[0, 8]$ is inserted. Fig. 2 (d) shows the SB-tree after inserting temporal data record 4.

The SB-tree has some drawbacks. In the environment where the same time points are not frequent, $O(\log N)$ disk updates are needed for one record insertion, where N is the number of temporal data records in the tree. This is because we have to update the values in the internal nodes on the path from the root node to a leaf node, and the height of the tree is proportional to $O(\log N)$. In addition, we have to store about $2N$ index records since each insertion usually generates two new index records.

3. THE PROPOSED METHOD

In this work, we use the *Corner* transformation and *column-scan* transformation techniques to make a scalar value for indexing time intervals. (We call the transformed value the *T-value*). The time intervals are transformed into T-values, and then constructed into a tree structure, called the *CTA-tree*.

3.1 T-value Generation using the Corner and Column-scan Transformation

The *T-value* generation process consists of the following two steps. In the first step, we map a time interval into a spatial point using the *Corner* transformation [10]. By the Corner transformation, a time interval $[t_s, t_e]$ is mapped to a spatial point (t_s, t_e) . In the transformed space, the way to find the time intervals that intersect with a given time interval is as follows: When determining whether two time intervals I_1 and I_2 intersect with each other, we compare the start time of one with the end time of the other. That is, the time intervals that intersect with the given time interval $[t_s, t_e]$ will have the start time less than or equal to t_e , and the end time greater than or equal to t_s .

In Fig. 3, we illustrate an example of a Corner-transformed time interval $I([3, 8])$, where p is the point that is transformed from I . Here, the time intervals that intersect with

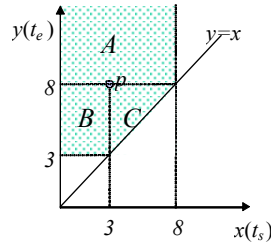


Fig. 3. The region of points that satisfy the given time interval.

I are represented by the region which is specified by $x \leq 8(I.t_e)$ and $y \geq 3(I.t_s)$. Also, since $I.t_e \geq I.t_s$, $y \geq x$. The points in A are specified by $x \leq 8$ and $y \geq 8$, those in B are by $x \leq 3$ and $3 \leq y \leq 8$, and those in C are by $3 \leq x \leq 8$, $3 \leq y \leq 8$ and $y \geq x$. Therefore, in order to find time intervals that intersect with the given time interval $I([3, 8])$, we have only to search the points located in A , B , and C (i.e., the shaded area in the figure).

In the second step, we transform the spatial point into a value, called *T-value*, using a linear transformation technique. There have been various linear transformation techniques like the *Z-ordering* and *Hilbert* curves that map the multidimensional data into one-dimensional data [11]. Among these transformation techniques, we use the column-scan transformation for simplicity. In the column-scan transformation, a spatial point is treated as a big number. For example, assume that the value of time is a two-digit number. Then, the T-value for the time interval of a temporal data record, $[30, 40]$, is 3040. (In all of the examples in the rest of paper, we use this simple transformation technique).

3.2 The Structure of the CTA-tree

The CTA-tree consists of two kinds of nodes: *A-node* and *I-node*. An *A-node* (*aggregate node*) stores the index records each of which consists of a T-value and the aggregate value for the T-value. An *I-node* (*index node*) indexes *A-nodes* or other *I-nodes*. An index record in the *I-node* consists of a T-value and a pointer to its child. The T-value in the *I-node* is the upper bound of the T-values in the child. The index records in the deepest *I-node* (e.g. I_1 and I_2 in Fig. 4 (d)) whose child nodes are *A-nodes* have one additional component, which is the largest end time among the time intervals in the *A-nodes* to which the *I-node* points.

Fig. 4 shows the construction steps of the CTA-tree for the example data in Fig. 1 (a). We describe the time points as two-digit numbers, and thus the T-values as four-digit numbers. That is, the T-values of the index records in Fig. 1 (a) are 0009, 0311, 0615, 1325, and so on. Here, we assume that the capacity values (i.e., the number of index records) for an *I-node* and an *A-node* are 2 and 3, respectively.

Fig. 4 (a) shows the CTA-tree after inserting temporal data record 1 and 2 in Fig. 1 (a). After inserting temporal data record 3 and 4, the CTA-tree becomes as in Fig. 4 (b). After inserting the data record 4 into the CTA-tree, A_1 splits into A_1 and A_2 , and then, I_1 becomes the new root. Now, we insert a new index record in I_1 . The T-value of the new index record is decided as if we split the x-axis of the 2-D space. Thus, the T-value becomes 0599 and A_2 contains the records whose T-values are greater than or equal to 0600.

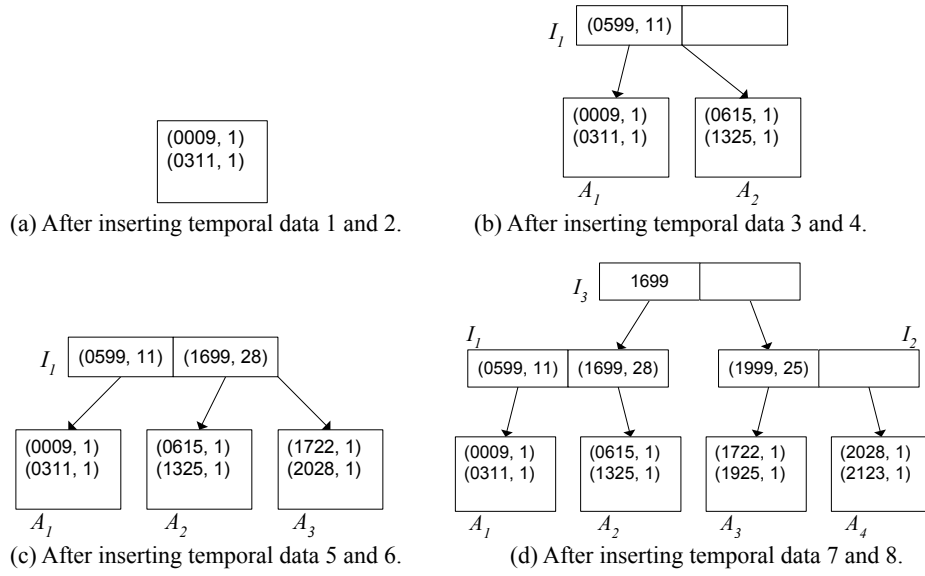


Fig. 4. The construction of the CTA-tree for COUNT.

Now, the largest end time of the index record is 11. This means that 11 is the largest end time of all the time intervals in A_1 . Fig. 4 (c) shows the CTA-tree after inserting data record 5 and 6. After inserting temporal data 7 and 8, the CTA-tree becomes as in Fig. 4 (d). In this process, A_3 splits into A_3 and A_4 , I_1 splits into I_1 and I_2 , and a new I -node I_3 becomes the new root.

When we insert a temporal data record in an A -node, some optimization can be considered for each of the following two aggregation classes:

- For the temporal aggregations of COUNT, SUM and AVG, we can merge two index records X_1 and X_2 into X_3 whose time interval is $[the\ start\ time\ of\ X_1, the\ end\ time\ of\ X_2]$ and the aggregate value is the same to that of X_1 , if the start time of X_2 is (the end time of $X_1 + 1$) and the aggregate value of X_1 is the same to that of X_2 .
 - ♦ Let us show an example based on the COUNT operation. Suppose two index records X_1 and X_2 whose intervals are $[3, 10]$ and $[10, 20]$, respectively. Also, there are no other index records that are relevant to these intervals. Then, since the count values of X_1 and X_2 are 1, and their intervals are contiguous, we can merge them into a new index record X_3 , where the count value of X_3 is 1 and the interval of X_3 is $[3, 20]$.
- For the temporal aggregations of the MAX and MIN, we can merge two index records X_1 and X_2 into X_1 if the time interval of X_1 covers the time interval of X_2 and the aggregate value of X_1 is greater (less for MIN) than or equal to that of X_2 .
 - ♦ Suppose two index records X_1 and X_2 , where their intervals and MAX values are $[5, 25]$ and $[10, 20]$, and 10 and 5, respectively. Since the time interval of X_1 includes that of X_2 and the MAX value of X_1 is greater than that of X_2 , X_1 and X_2 can be merged into X_1 .

3.3 Aggregation Processing on the CTA-tree

The condition phrase of a temporal aggregation consists of a time interval I that consists of a *start time* t_s and an *end time* t_e . In order to process the temporal aggregation, we visit all the paths of I -nodes that satisfy the given condition. To find the paths, we use the T-values and the largest end times in I -nodes. The T-values in the I -nodes are the upper bounds of T-value ranges. The time intervals that intersect with the given time interval I are represented by the T-values which are formed as ' $\alpha\beta$ ' where $\alpha \leq I.t_e$ and $\beta \geq I.t_s$. The query area that contains these T-values is a part of rectangle as shown in Fig. 3. (We can treat the query area as a rectangle since there are no data below the $y = x$ line). We visit a child if the T-value range intersects with the query area. We use the largest end times in the deepest I -nodes to decide whether we have to visit child nodes or not. Although the T-value of an index record in the deepest I -node intersects the query area, we need not visit the child node if the largest end time is less than the start time of the given time interval. This is because there are no index records whose time intervals intersect with the given time interval.

While visiting the nodes, partial results of aggregation are gathered from A -nodes. And we can compute the total result by merging these partial results. For example, assume that the time interval is $[1, 9]$ in the given condition. Then, in Fig. 4 (d), we visit the paths $I_3-I_1-A_1$ and $I_3-I_1-A_2$. The partial results from the above two paths are $\{([1, 2], 1), ([3, 9], 2)\}$ and $\{([6, 9], 1)\}$. Those results can be described as $\{([1], 1), ([3], 2), ([10], 0)\}$ and $\{([6], 1), ([10], 0)\}$, since the constant intervals can be expressed as a sequence of start times. By merging these results, the total result can be computed as $\{([1], 1), ([3], 2), ([6], 1), ([10], 0)\}$.

4. ANALYSES AND EXPERIMENTS

4.1 Complexity Analyses

In this section, we analyze the number of disk updates required for one data record insertion and the number of index records in the tree after inserting N records.

In the SB-tree, we need $2h - 1$ disk updates for one record insertion in the worst case, where h is the height of the tree. This worst case occurs when the following two conditions are satisfied in an I -node: (1) the time interval of the inserted record covers at least one time interval and intersects two time intervals, and (2) the aggregate value for the time interval covered by the inserted time interval is changed due to the insertion. In this case, the internal node must be updated so as to store the changed aggregate value. And, we must visit the child nodes of the two index records whose time intervals intersect with the inserted time interval. This process is repeated for the two paths starting from the root node. Because the number of nodes of each path is $(h - 1)$ except the root node, we need $2h - 1$ updates for one data record insertion.

In the CTA-tree, an insertion of temporal data record requires only two disk updates in the worst case. We store a new index record in an A -node *i.e.*, we update the A -node. The index records in the deepest I -nodes have the largest end times, and these largest end times are changed when the new data record has an end time greater than them. For one data record insertion, we update the index record in the deepest I -node whose child node

is the A -node into which we insert the new data record. Therefore, we need only two disk updates even in the worst case, one for the A -node and one for the deepest I -node. In the case of node splits, the update cost of the CTA-tree is equal to that of the SB-tree. However, since most insertions are processed by only node updates, the insertions causing node splits are not frequent.

In the SB-tree, two index records can be generated for one record insertion. When we insert a temporal data record, two time points, ‘the start time-1’ and ‘the end time’, are placed in the A -nodes if necessary. Therefore, in the worst case, there can be totally $2N$ index records in the tree for N temporal data records. In the CTA-tree, at most one index record will be generated for one data record insertion. Therefore, totally N index records can be generated in the worst case.

Table 1. The worst case analysis of storage and update cost.

	The number of disk updates required for one record insertion	The number of index records in the tree for storing N temporal data records
SB-tree	$2h - 1$	$2N$
CTA-tree	2	N

4.2 Performance Experiments

We experimentally compared the performance of the SB-tree and the CTA-tree with respect to (i) the storage cost, (ii) the update cost and (iii) the aggregation processing time. For the experiments, we generated a set of temporal data records, and construct the SB-tree and CTA-tree for the data set. We assumed that all attributes have non-negative values. The experimental parameters are the number of data records and the density. The numbers of data records are 10000, 20000, 50000, 100000, 200000, 500000 and 1000000. We assumed that the time interval has a uniform distribution. The density is the ratio of unique time points to the number of all time points. The density values are 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20%. The length of time intervals varies according to the numbers of temporal data and density values. All attributes and pointers are 4 bytes in size.

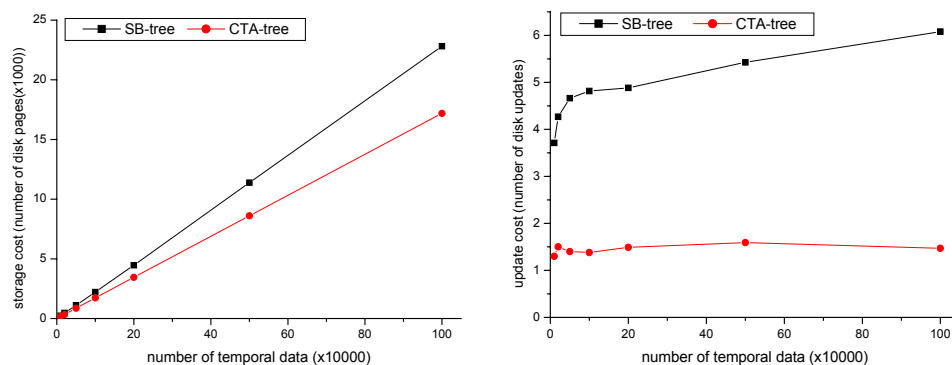


Fig. 5. The storage and update cost for various numbers of temporal data.

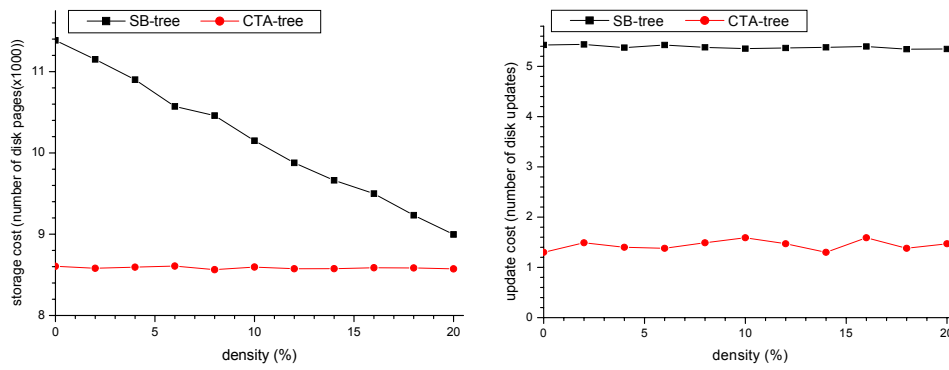


Fig. 6. The storage and update cost for various densities.

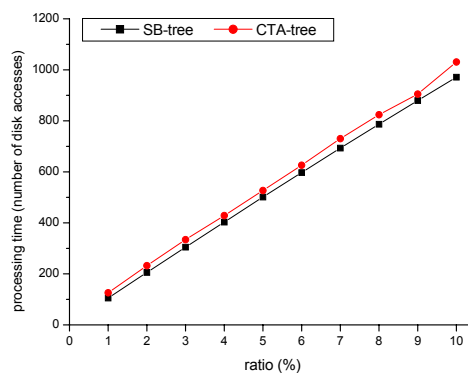


Fig. 7. The aggregation processing time.

Firstly, in Fig. 5, we examine the disk usage and the number of disk updates for one temporal data record insertion. The figure shows that the CTA-tree uses less space than the SB-tree. The disk usage of the CTA-tree is about 2/3 of that of the SB-tree when the number of temporal data is 1000000. Also, it shows that the CTA-tree updates less disk pages than the SB-tree. The number of disk updates of the CTA-tree is about 1/4 of that of the SB-tree when the number of data is 1000000.

Fig. 6 shows the disk usage and the number of disk updates for various densities. The number of temporal data records is 500000. Although the number of disk pages of the SB-tree decreases as the density becomes higher, the CTA-tree uses still less. Also, it is observed that the number of disk updates does not change in both the trees, and that of the CTA-tree is about 1/4 of that of the SB-tree.

Fig. 7 shows the number of disk accesses for processing temporal aggregations. We set the number of data as 500000 and the density as 10%. We change the length of the time interval as 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10% of the entire time length. Although the SB-tree is slightly superior to the CTA-tree, the difference is very little.

As the size of the TDB increases, the amount of data for the temporal aggregation increases. Also, many update-intensive applications make frequent insertions into the temporal databases. For example, in the applications like LBS (Location Based Service),

the number of mobile phones in a cell changes very frequently. These changes produce lots of insertions of new temporal records into the database. In these update-intensive environments, the proposed method shows good performance behaviors – the low storage overhead, the low insertion overhead, and the similar aggregation processing performance compared with the SB-tree.

5. CONCLUSIONS

Aggregate functions calculate a value or select a representative value from a (part of) relation. They are essential to many applications. But there are many differences between the conventional aggregation and the temporal aggregation that supports the time dimension. Thus, techniques for the conventional aggregation are not easy to be used directly for the temporal aggregation.

There have been several proposals to compute temporal aggregates. Among them, the SB-tree is known to be the most efficient [5]. However, the SB-tree is not efficient in highly update-intensive environments, since the SB-tree requires $O(\log N)$ disk updates for one temporal data insertion, and $2N$ index records are generated in the tree when we insert N temporal data.

In the paper, we proposed a new tree structure called the *CTA-tree* and an aggregate processing method based on the CTA-tree. We used two transformation techniques (the *Corner* transformation and *column-scan* transformation) to make a value for indexing, and store the index records based on those values. The CTA-tree supports efficient insertions of temporal data records and effective query processing. Through analyses and experiments, we evaluated the aggregation processing performance, update efficiency and storage cost of the proposed and the conventional methods. The results showed that our CTA-tree has better storage utilization and needs fewer disk updates than the SB-tree without sacrifice of aggregation processing efficiency.

REFERENCES

1. C. S. Jensen and R. T. Snodgrass, "Temporal data management," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, 1999, pp. 36-44.
2. J. S. Kim and M. H. Kim, "An effective data clustering measure for temporal selection and projection queries," *Decision Support Systems*, Vol. 30, 2000, pp. 33-50.
3. E. Zimanyi, "Temporal aggregates and temporal universal quantification in standard SQL," *SIGMOD Record*, Vol. 35, 2006, pp. 16-21.
4. J. S. Kim, S. T. Kang, and M. H. Kim, "On temporal aggregate processing based on time points," *Information Processing Letters*, Vol. 71, 1999, pp. 213-220.
5. D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger, "Efficient computation of temporal aggregates with range predicates," *Principles of Database Systems*, 2001, pp. 237-245.
6. D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger, "Temporal and spatio-temporal aggregations over data streams using multiple time granularities," *Information Systems*, Vol. 28, 2003, pp. 61-84.
7. J. Yang and J. Widom, "Incremental computation and maintenance of temporal ag-

- gregates,” *VLDB Journal*, Vol. 12, 2003, pp. 262-283.
8. B. Moon, I. F. V. Lopez, and V. Immanuel, “Efficient algorithms for large temporal aggregation,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, 2003, pp. 744-759.
 9. D. Gao, J. Alving, and G. Gendrano, “Main memory-based algorithms for efficient parallel aggregation for temporal databases,” *Distributed and Parallel Databases*, Vol. 16, 2004, pp. 123-163.
 10. J. W. Song, K. Y. Whang, Y. K. Lee, M. J. Lee, and S. W. Kim, “Spatial join processing using corner transformation,” *IEEE Transactions of Knowledge and Data Engineering*, Vol. 11, 1999, pp. 688-695.
 11. V. Gaede and O. Gunther, “Multidimensional access methods,” *ACM Computing Surveys*, Vol. 30, 1998, pp. 170-231.

Sung Tak Kang received his Ph.D. and M.S. degrees in Computer Science from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 2004 and 1998, respectively, and his B.E. degree in Computer Science from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1996. His research interests include temporal databases, data warehouses and online analytical processing.

Yon Dohn Chung is an associate professor in the department of Computer Science and Engineering, Korea University, Seoul, Korea. He received his B.S. degree in Computer Science from the Korea University in 1994, and his M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1996 and 2000, respectively. His research interests include mobile databases, data processing over wireless/mobile/sensor networks and distributed systems. He is a member of the ACM and IEEE.

Myoung Ho Kim received his B.S. and M.S. degrees in Computer Engineering from the Seoul National University, Seoul Korea, in 1982 and 1984, respectively, and his Ph.D. degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. In 1989, he joined the faculty of the Department of Computer Science at KAIST, Daejeon, Korea, where currently he is a professor. His research interests include database systems, data stream processing, sensor networks, workflow, XML and distributed processing. He is a member of the ACM and IEEE Computer Society.