Comput Syst Sci & Eng (2004) 5: 3–9
© 2004 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

Transformation-based temporal aggregation using order-based buffer replacement strategy

Joon-Ho Woo*, Byung Suk Lee†, Min-Jae Lee*, Jae-Gil Lee* and Kyu-Young Whang*

*Department of Computer Science and Advanced Information Technology Research Center (AITrc),
Korea Advanced Institute of Science and Technology (KAIST). Email: {jhwoo, mjlee, jglee, kywhang}@mozart.kaist.ac.kr
†Department of Computer Science, University of Vermont, VT, USA. Email: bslee@cs.uvm.edu

We present a new method for computing temporal aggregation based on dimension transformation. The novelty of our method lies in transforming the start time and end time of one-dimensional temporal tuples to two-dimensional data points and storing the points in a two-dimensional index. It then calculates temporal aggregates through a temporal join between the data in the index and the base intervals (defined as the intervals delimited by the start time or end time 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 further improve the efficiency with a new buffer page replacement technique that predicts the page access order within the index. We demonstrate the efficacy of our method through experiments.

Keywords: temporal databases, temporal aggregation, multi-dimensional indexes, buffer management

1. INTRODUCTION

A temporal database is essential to applications dealing with time-varying data [11], such as trend analysis and prediction in decision support systems, version management in computer aided designs, medical record management, and data warehousing. In this regard, a temporal database system is used to store and query time-varying information.

There has been extensive research on temporal databases [25]. The early research focused mainly on conceptual problems like data models and query languages. An example is the temporal SQL(TSQL)[19], which extends the relational data model and the query language SQL with temporal features. Subsequently, the research shifted toward system implementation problems like query processing in which query operations have been extended to support temporal data models and queries. Among various query operations, temporal aggregation is particularly important due to the high processing cost [13].

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* [13], and performs the aggregation in each constant interval. There are typically two kinds of aggregate functions: cumulative (e.g. COUNT, SUM, AVG) and selective (e.g. MIN, MAX). Aggregation is an expensive operation, and temporal aggregation is particularly more so because it deals with very large data accumulated over time and has to calculate aggregate values separately for multiple time intervals.

There have been several temporal aggregation methods proposed to date. The early ones include the aggregation tree method [13] and its variants [24, 8, 12, 15]. Although these methods do facilitate calculating temporal aggregates, they require one aggregation tree for each aggregate function. Moreover, they require the tree structures to reside in main memory. The trees, however, are typically much larger than the available main memory because a temporal database

retains all tuples from the past.

Yang and Widom [26] have proposed a method using the SB-tree, which is disk resident. 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 pre-calculated aggregate values. However, the overhead of immediate updates is nontrivial, particularly for selective aggregate functions. For example, if the deleted tuple is the minimum one, then the method requires re-reading *all* tuples in the time interval of the deleted tuple in order to calculate a new minimum. Moreover, the update should be done for each aggregate function through its own SB-tree. Thus, this method is suitable only in an environment with infrequent insertions, deletions, or updates of tuples and relatively more frequent queries.

Another problem is that the SB-tree is not usable for queries predicated on ordinary (i.e. non-temporal) attributes. An example query is, "*find the average salary of all employees hired between JAN-01-2000 and DEC-31-2002 and currently working in the sales department*." Zhang *et al.* [27] proposed a new method to solve this problem. The method, however, is restricted to only the *transaction time* dimension and only the COUNT, SUM and AVG aggregate functions.

In this paper, we propose a new method based on two-dimensional transformation. In this method, the start time and end time of a temporal tuple are transformed to a data point in a two-dimensional space, and the data point is stored and retrieved through a multi-dimensional index. It calculates the aggregates through a temporal join between the data in the index and the base interval constituting the constant interval. This calculation is done by incrementally modifying the aggregate from the previous base interval without re-reading all tuples for the current base interval.

Compared with the aggregation tree methods, our method requires only one index tree for all aggregate functions and does not require the tree structure to be resident in main memory. Compared with the SB-tree method, our method does not incur any extra overhead when updating tuples and, therefore, is significantly more efficient for update operations. Besides, it uses only one index structure for all possible aggregate functions and leverages multi-dimensional indexes common in spatial or spatio-temporal databases. Furthermore, it can process the temporal aggregations with a predicate on ordinary attributes by just adding dimensions for those attributes in the multi-dimensional index. Unlike the method proposed by Zhang *et al.* [27], our method can process any other time dimension as well as transaction time dimension.

Since the index structure is disk-resident in our method, the buffer management is crucial for efficient aggregation computation. The conventional strategy like the Least-Recently-Used (LRU) may be appropriate if the order of page accesses is unpredictable [3], but the performance will be better if the order can be predicted. This paper describes how to predict the order and proposes a new order-based buffer page replacement technique. Experimental results show that our technique reduces the elapsed time and the number of disk page accesses as much as 3.8 times over the LRU technique.

Following this introduction, Section 2 provides some background information. Section 3 describes related work. Section 4 describes the proposed temporal aggregation method. Section 5 describes the order-based buffer management technique, and Section 6 compares the performance with the case where the LRU buffer replacement technique is used. Finally, Section 7 concludes the paper.

2. BACKGROUND

In this section we briefly review some relevant concepts of temporal aggregation in Section 2.1, and introduce the multi-dimensional index used in our work in Section 2.2.

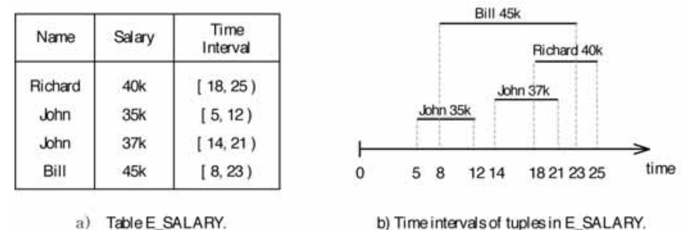
2.1 Temporal aggregations

Each tuple in a temporal relation has an associated time interval [13]. This time interval can be determined based on an instant on the time axis (called the *instant grouping*) or based on a fixed span like the year, month, week, or day (called the *span grouping*) [15]. There are constant intervals defined in both cases. Because the span grouping can be regarded a special case of the instant grouping, we focus on the instant grouping in this paper.

Figure 1a shows an example 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.[1] Figure 2 shows the COUNT and MIN of *Salary* changing over time. Note that each aggregate function generates different constant intervals. For example, for COUNT, the constant intervals are [5, 8), [8, 12), [12, 14), [14, 18), [18, 21), [21, 23), [23, 25); for MIN, they are [5, 12), [12, 14), [14, 21), [21, 25).

2.2 Multi-dimensional index

Garcia-Molina *et al.* [6] categorized multi-dimensional index structures into "hash-like" [16, 2] and "tree-like" [1, 5, 9]. In our work we use the multilevel grid file (MLGF) [22],[2] which is both hash-like and tree-like. It is a multilevel extension of the grid file and is similar to the K-D-B-tree [18] – a disk version of the K-D-tree – but, uses hashing. Specifically, it is a dynamic hash file supporting multi-key accesses to data through a multilevel directory tree structure. We particularly use the minimum bounding rectangle (MBR)-MLGF



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

a) Table E_SALARY.

Figure 1 An example of temporal relation. (a) Table E_SALARY, (b) time intervals of tuples in E-SALARY

1 We represent the time as an integer for convenience without loss of generality.
2 In principle, any multi-dimensional point access method [7] can be used for the same purpose.

Constant Interval	COUNT
[5, 8)	1
[8, 12)	2
[12, 14)	1
[14, 18)	2
[18, 21)	3
[21, 23)	2
[23, 25)	1

a) COUNT.

Constant Interval	MIN
[5, 12)	35k
[12, 14)	45k
[14, 21)	37k
[21, 25)	40k

b) MIN.

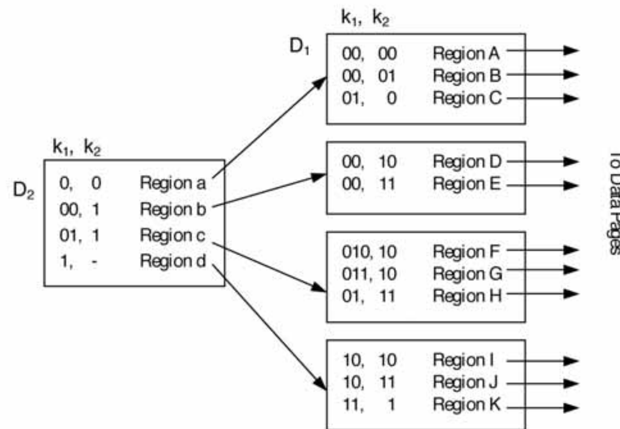
Figure 2 Temporal aggregates COUNT and MIN on E_SALARY.Salary at their constant intervals.

[20]. The MBR-MLGF is an extension of the MLGF targeted toward efficient *spatial* or *spatio-temporal* data accesses. In this section, we describe the structure of the MLGF and its extension to the MBR-MLGF.

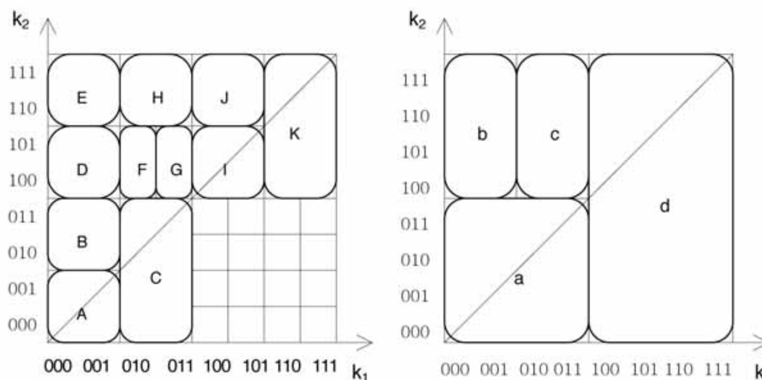
An MLGF is made of a multi-level directory structure and data pages. Each directory level reflects partitioned space, and each directory entry consists of a region vector and a pointer to either a data page or the lower-level directory page. A region vector consists of hashing keys, one key per dimension. The *i*th hashing key value of a region vector is the prefix common to the *i*th hashing key values of all records in the region. The region of a directory entry encompasses the regions of all directory entries under it.

MLGF adapts to dynamic changes of data by splitting or merging pages as data are inserted or deleted, respectively. When a data record is inserted into the data space, the record's keys are hashed to find the region the record belongs to, and the record is inserted into the data page allocated to the region. If an overflow occurs as a result, the region is split into halves, a new page is allocated, and part of the data are moved into the new page. MLGF uses local splitting strategy [22], by which it splits only the necessary regions instead of the hyperplane of the entire file. This strategy allows for keeping only the minimum required directory entries. As a result, the size of an MLGF directory is determined by only the number of inserted data records and is independent of the distributions or the correlation among the attributes of the data [14]. In this regard, MLGF is suitable for handling temporal data because there exists high correlation between the start time and the end time of a temporal tuple as evidenced by the constraint (start time < end time).

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 eleven entries in D_1 – one for each of the eleven regions *A*



a) A two-level directory structure.



b) Regions in D_1.

c) Regions in D_2.

Figure 3 An example MLGF

through K, and 4 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 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 [20], 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 $min\text{-}t_s$ and horizontal line at $max\text{-}t_e$ reduces R_1 to its MBR.

3. RELATED WORK

Tuma [21] proposed a two-step algorithm for computing temporal aggregates, where each step requires a full database scan. The first step finds the intervals of the aggregate result tuples using a linked-list, and the second step updates the values of all result tuples affected by each tuple. This method takes O(mn) time, where m is the number of result tuples and n is the number of tuples.

Kline and Snodgras [13] proposed the aggregation tree based on the binary segment-tree [17]. The segment-tree feature allows efficient processing of tuples with long intervals. Each node of the tree contains the tuple's lifespan and the value used for aggregation. Children nodes' time intervals are mutually exclusive and exhaustive partitions of the parent's time interval, and a leaf node's time interval makes one constant interval. The aggregation result of one constant interval is obtained by reading all node from the root node to the leaf node, and the entire result is obtained by depth-first-search of the tree.

One drawback of the aggregation tree is that it is a main-memory data structure, which limits its effectiveness as a data structure for maintaining temporal aggregates. Another drawback is that it is unbalanced. In the worst case, it takes O(n^2) to compute a temporal aggregate from a table with n tuples. There have been balanced aggregation trees [12, 15] proposed so that the worst-case run time is O(n log m). These trees are, however, still main-memory data structures. In addition, there have been parallel versions of the aggregation-trees [24, 8], but they all inherit the same limitations of the sequential versions.

Yang and Widom [26] proposed the SB-tree, which is similar to the aggregation tree in terms of storing a time interval and a value in each node. However, being based on the B-tree [4], the SB-tree is different in that it stores *multiple* time intervals in each node, is balanced, and is disk-resident. As mentioned in Introduction, the SB-tree incurs non-trivial update cost of a tuple. Moreover, the SB-tree is not usable for temporal aggregation predicated on ordinary attributes.

Zhang *et al.* [27] proposed the Multi-Version SB-tree (MVSB-tree) for temporal aggregations with predicates on ordinary attributes. The MVSB-tree is a forest of SB-trees, and the root node of each SB-tree is stored in a structure like the B-tree. The MVSB-tree stores data with non-decreasing time trend, and the aggregation result is obtained using a prefix-sum technique. So, it cannot be applied to data with a *valid time* dimension and can process only the COUNT, SUM, and AVG aggregate functions.

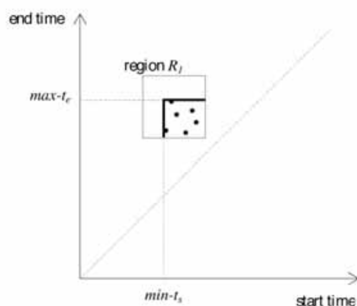


Figure 4 Minimum bounding region in an MBR-MLGF

4. THE PROPOSED TEMPORAL AGGREGATION METHOD

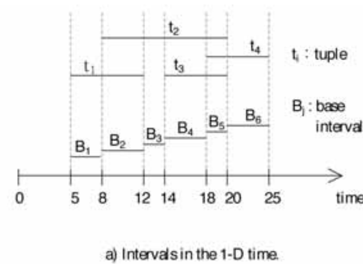
As mentioned in Introduction, we represent temporal tuples as points in a two-dimensional (2-D) space defined by the start time and end time of the tuples. Based on this concept, we provide the notion of a temporal join window in Section 4.1 and present the aggregation algorithms in Section 4.2.

4.1 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



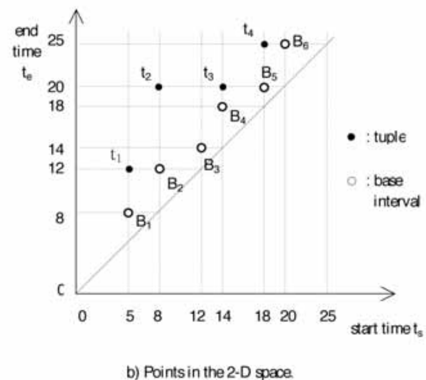


Figure 5 Tuples and base intervals

merged intervals make one constant interval [13] 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 5a shows the time intervals of four tuples and six base intervals of COUNT, and Figure 5b shows the 2-D points mapped from these intervals.

Given the 2-D representation of time intervals, the temporal join window is defined as follows, in a way similar to the spatial join window proposed by Song *et al*. [20].

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

$$TJW(I_i) = \{< t_s, t_e > \mid t_s < e_i \text{ and } t_e > s_i\}$$

If the interval is a base interval, then by definition there cannot be the start time or end time of any tuple within the interval, and hence, Definition 2 is refined as follows.

Definition 3 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 $B_i \equiv [s_i, e_i)$,

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

Figure 6 shows the temporal join window (TJW) for a base interval B_i. In Figure 5b, for example, $TJW(B_4)$ 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' 2-D points are located on the *grid points* formed by intersecting the boundaries of the TJWs of base intervals. Formally,

Property 1 TJW grid
Consider all base intervals $B_i \equiv [s_i, e_i)$, i = 1,2 ... of the given temporal tuples. Then, for each tuple $[t_s, t_e)$, there exist B_j such that $t_s = s_j$ and B_k such that $t_e = e_k$.

Proof
Straightforward from the definition of the base interval.

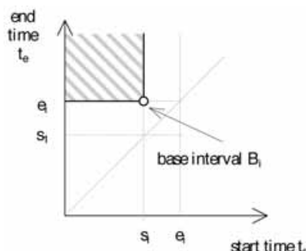


Figure 6 Temporal join window TJW(B_i) for a base interval B_i

4.2 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 7, 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 calculating 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 7, the 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 the tuples' points are located only at the grid points formed by the TJWs (see Property 1), all tuples in A have the end time e_{i-1} (while all tuples in B+C have the end time e_i) and all tuples in the region C+E have the start time s_i (while all tuples in F have the start time s_{i+1}). Let $N_{s(s_i)}$ be the number of tuples with the start time s_i and $N_{e(e_i)}$ 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

$$\text{COUNT}(B_i) = \text{COUNT}(B_{i-1}) - N_e(e_{i-1}) + N_s(s_i) \quad (1)$$

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 between the MIN in B+D (= MIN in B_{i-1}) and the MIN in C+E. In case the MIN in B_{i-1} and the MIN in A are the same, the tuple with the minimum value must be in A and, therefore, the MIN in B_i is the smaller between the MIN in A and the MIN 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

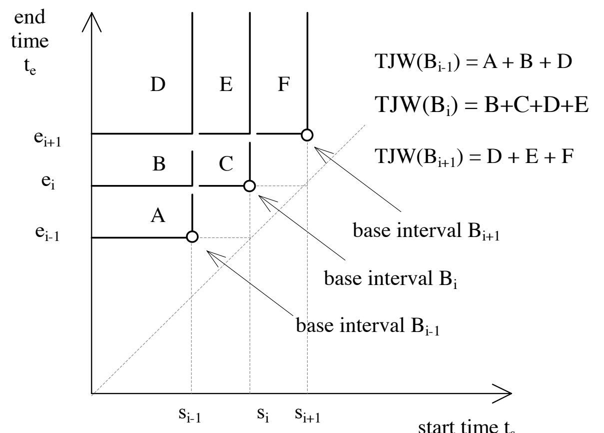


Figure 7 Temporal join windows of three consecutive base intervals.

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

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

5. ORDER-BASED BUFFER-PAGE REPLACEMENT

Since the multi-dimensional index is disk-resident, buffer management is important to reduce the number of pages fetched from the disk, and its performance is influenced significantly by the page replacement strategy [23]. The conventional strategy like the Least-Recently-Used (LRU), is appropriate if the order of page accesses is unpredictable [3]. A salient feature of our method, however, is that it can predict the order of page access. As mentioned in Section 4.2, temporal aggregation is performed in the order of the base intervals for efficiency's sake. Our page replacement strategy takes advantage of this order.

Let us explain our order-based page replacement strategy using Figure 8. To calculate an aggregate for the base interval B_i, we need to read the pages containing tuples with the start time s_i (e.g. s_2, s_3, s_4) and the pages containing tuples with the end time e_{i-1} (e.g. R_2, R_3, R_4, R_5, R_6). (See the algorithms for COUNT and MIN in Section 4.2.) Therefore, the disk access performance can be improved by retaining these pages in the buffer.

If the buffer overflows while aggregating for B_i, a victim page must be chosen to be replaced. Among the pages read for the aggregates up to B_{i-1}, the pages below TJW(B_{i-1}) (e.g. R_1) are never needed again as the aggregation progresses in the order of the base intervals B_i, B_{i+1}, etc. Therefore, they are the first candidates of the victim. Note that the pages whose maximum end time is e_{i-1} (e.g. R_2) are needed for the aggregate for B_i but, once used, would become the first candidates of the victim for the aggregate for B_{i+1}. The pages at the top of TJW(B_i) (e.g. T, S_1) are needed only when the last base interval is processed and, therefore, are the second candidates.

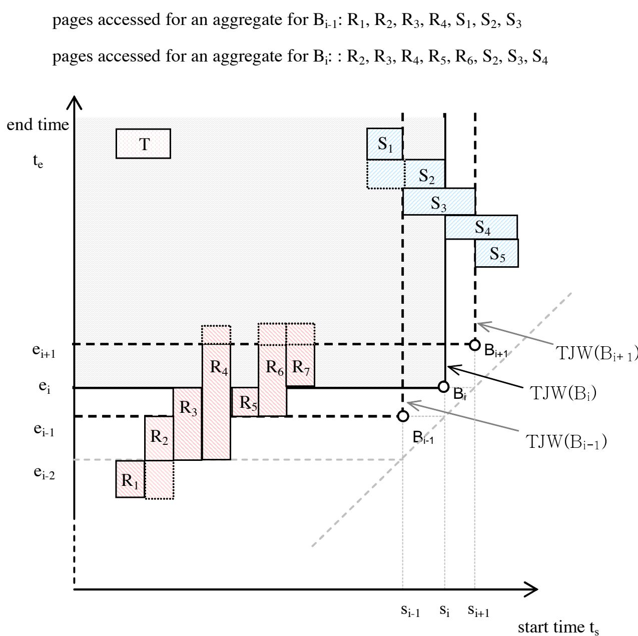


Figure 8 Buffer page management for aggregate for B_i

Figure 9 shows Algorithm *Pick_Victim*, which implements the page replacement strategy presented here. To pick the victim page to be replaced, it first looks for a page at a location below TJW(B_i) (Line 4). If no such page exists, it searches TJW(B_i) for a page at the top, whose smallest end time of the tuples is the largest among all pages (Lines 7–11).

6. PERFORMANCE EVALUATION

We have conducted experiments to evaluate our buffer page replacement algorithm (see Figure 9). In this section we describe the experimental setup in Section 6.1 and present the results in Section 6.2.

6.1 Experimental setup

Experimental data

We use two synthetic data sets (DS1, DS2) generated in a way similar to the data used by Kline and Snodgrass [13] and Moon, Lopez, and Immanuel [15]. 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 65536, 262144, 1048576, and 4194304 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, etc. (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 1/8 and 1/8 of the time range for the span (= end time – start time).

Performance metric

We use the elapsed time and the number of disk page accesses as the cost metrics, and measure the performance as the sum of the costs of five aggregate functions COUNT, SUM, AVG, MIN, and MAX. To avoid noise, we execute each function more than three times and calculate an average.

System configuration

The system is configured in Linux Server with 512 Mbyte RAM and ATA-3 IDE hard disk drive, and 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. We implement the algorithms using C/C++ language.

Algorithm Pick_Victim
Input:
- PageTable: a page mapping table (in which each page entry contains the minimum end time (min_t_e) and the maximum start time (max_t_s) among all tuples in the page)
- B_i: constant interval $[s_i, e_i)$

Output:
- victim: the victim page

begin
1: m_t_e := 0; /* track the smallest end time of tuples */
2: $e_{i-1} = s_i$; /* the end time of the previous base interval B_{i-1} */
3: for each page entry p in PageTable {
4 if (p.max_t_e < e_{i-1}) /* the page is below TJW(B_{i-1}). */
5: return p;
6: else
7: if (p.min_t_e > e_i and p.max_t_s < s_i) /* the page is in TJW(B_i) */
8: if (p.min_t_e > m_t_e) { /* the pageís min_t_e is the largest so far */
9: victim := p;
10: m_t_e := p.min_t_e;
11: }
12: }
13: return victim;
end

Figure 9 Victim page selection algorithm

6.2 Experimental result

We compare our *Pick_Victim* algorithm (shown in Figure 9) with the LRU algorithm by executing the aggregate queries while varying the buffer size. Figures 10 and 11 respectively show the performance obtained using the data sets DS1 and DS2 of size 1 Mbytes. The results show that our algorithm is up to 3.8 times more efficient than the LRU, especially for smaller buffer sizes. We have tested with other relation sizes as well, and the results show very similar trend.

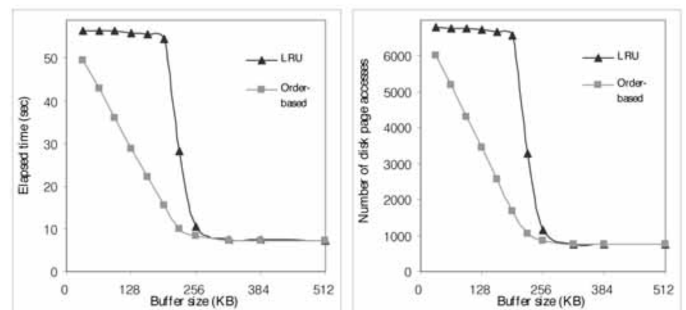


Figure 10 Performance comparison of the buffer replacement algorithms using DS1 of size 21 Mbytes

7. CONCLUSIONS

We have presented a new temporal aggregation method. It transforms temporal tuples to two-dimensional points and stores them through a two-dimensional index. Then, it calculates the aggregates by identifying the temporal join window (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 re-reading 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 improved the aggregation efficiency further by taking advantage of the order of base intervals in the buffer page replacement. Experiments show that our buffer page replacement algorithm is superior to the conventional LRU algorithm, especially for small buffers.

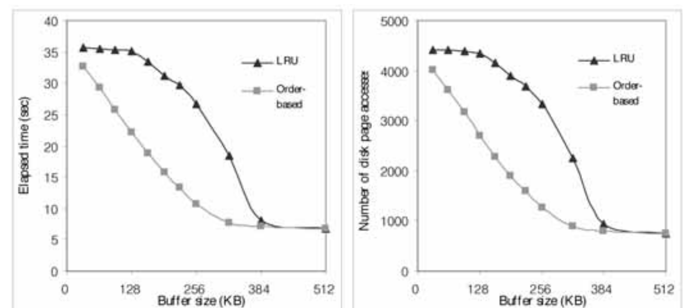


Figure 11 Performance comparison of the buffer replacement algorithms using DS2 of size 1 Mbytes

ACKNOWLEDGEMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

REFERENCES

1 **J. L. Bentley** Multi-dimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM*, Vol. 18, No. 9, pp. 509–517, 1975.

2 **W.A. Burkhard** Hashing and Trie Algorithm for Partial Match Retrieval, *ACM Trans. on Database Systems*, Vol. 1, No. 2, pp. 175–187, 1976.

3 **E. G. Coffman Jr. and P. J. Denning** *Operating Systems Theory*, Prentice-Hall, 1973.

4 **D. Comer** The Ubiquitous B-Tree, *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121–137, 1979.

5 **R. A. Finkel and J. L. Bentley** Quad Trees, a Data Structure for Retrieval on Composite Keys, *ACM Trans. on Database Systems*, Vol. 1, No. 2, pp. 175–187.

6 **H. Garcia-Molina, J. D. Ullman, and J. Widom** *Database Systems: The Complete Book*, Prentice Hall, 2002.

7 **Volker Gaede and Oliver Gunther** Multidimensional Access Methods, *ACM Computing Surveys*, Vol. 30, No. 2, pp. 170–231, 1998.

8 **J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass** Parallel Algorithms for Computing Temporal Aggregates, In *Proc. IEEE Int'l Conf. on Data Engineering*, Sydney, Austalia, pp. 418–427, 1999.

9 **A. Guttman** R-trees: a Dynamic Index Structure for Spatial Searching. In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Boston, USA, pp. 47–57, 1984.

10 **C. S. Jensen, C. E. Dyreson, M. H. Bohlen *et al.*** The Consensus Glossary of Temporal Database Concepts – February 1998 Version, Temporal Databases: Research and Practice, *Lecture Notes in Computer Science*, Vol. 1399, Springer-Verlag, ISBN 3-540-64519-5, pp. 367–405, 1998.

11 **C. S. Jensen and R. T. Snodgrass** Temporal Data Management, *IEEE Transacions on Knowledge and Data Engineering*, Vol. 11, No. 1, pp. 36–44, 1999.

12 **J. Kim, S. Kang, and M. Kim** Effective Temporal Aggregation Using Point-Based Trees, In *Proc. Int'l Conf. on Database and Expert Systems*, Florence, Italy, pp. 1018–1030, 1999.

13 **N. Kline and R. T. Snodgrass** Computing Temporal Aggregates, In *Proc. IEEE Int'l Conf. on Data Engineering*, Taipei, Taiwan, pp. 222–231, 1995.

14 **S. Kim and K. Whang** Asymptotic Directory Growth of the Multilevel Grid File, In *Proc. Int'l Syp. on Next Generation Database Systems and Their Applications*, Fukuoka, Japan, pp. 257–264, 1993.

15 **B. Moon, I. F. V. Lopez and V. Immanuel** Scalable Algorithms for Large Temporal Aggregation, In Proc. IEEE Int'l Conf. on Data Engineering, San Diego, USA, pp. 145–154, 2000.

16 **J. Nievergelt, H. Hinterberger and K. Sevcik** The Grid File: An Adaptable, Symmetric, Multikey File Structure, *ACM Trans. on Database Systems*, Vol. 9, No. 1, pp. 38–71, 1984.

17 **F. P. Preparata and M. I. Shamos** *Computational Geometry: An Introduction*, Springer-Verlag, Berlin/Heidelberg, Germany, 1985.

18 **J. T. Robinson** The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes, In *Proc. Int'l Conf. on Management of Data*, *ACM SIGMOD,* New York, USA, pp. 10–18, 1981.

19 **R. T. Snodgrass *et al.*** *The TSQL2 Temporal Query Language*, Kluwer, 1995.

20 **J. Song, K. Whang, Y. Lee, M. Lee and S. Kim** Spatial Join Processing Using Corner Transformation, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 11, No. 4, pp. 688–698, 1999.

21 **P. A. Tuma** Implementing Historical Aggregates in TempIS, *Master's Thesis*, Wayne State Univ., 1992.

22 **K. Whang and R. Krishnamurthy** Multilevel Grid Files, *IBM Research Report* RC11516, 1985.

23 **K. Whang and R. Krishnamurthy** Query Optimization in a Memory-Resident Domain Relational Calculus Database System, *ACM Trans. on Database Systems*, Vol. 15, No. 1, pp. 67–95, 1990.

24 **X. Ye and J. A. Keane** Processing temporal aggregates in parallel, In *Proc. IEEE Int'l Conf. on Systems, Man, and Cybernetics*, Orlando, USA, pp. 1373–1378, 1997.

25 **Y. Wu, S. Jajodia and X. S. Wang** Temporal Database Bibliography Update, In *Proc. Int'l Conf. on Temporal Databases,* Dagstuhl, Germany, 1997.

26 **J. Yang and J. Widom** Incremental Computation and Maintenance of Temporal Aggregates, in *Proc. IEEE Int'l Conf. on Data Engineering*, Heidelberg, Germany, pp. 51–60, 2001.

27 **D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos and B. Seeger** Efficient Computation of Temporal Aggregate with Range Predicates, *ACM Symp. on Principles of Database Systems*, Santa Barbara, USA, 2001.