TATS: an Efficient Technique for Computing Temporal Aggregates for Data Warehousing

Young-Ok Shin, Sung-Kong Park, Doo-Kwon Baik, and Keun-Ho Ryu

An important use of data warehousing is to provide temporal views over the history of source data. It is significant that nearly all data warehouses are dependent on relational database technology, yet relational databases provide little or no real support for temporal data. Therefore, it is difficult to obtain accurate information for time-varying data. In this paper, we are going to design a temporal data warehouse to support time-varying data efficiently. For this purpose, we present a method to support temporal query by combining a temporal query process layer with the relational database which is used as a source database in an existing data warehouse. We introduce the Temporal Aggregate Tree Strategy (TATS), and suggest its algorithm for the way to aggregate the time-varying data that is changed by the time when the temporal view is created. In addition, The TATS and the materialized view creation method of the existing data warehouse have been evaluated. As a result, the TATS reduces the size of the fact table and it shows a good performance for the comparison factor in case of processing the query for time-varying data.

Manuscript received August 30, 1999 ; revised July 18, 2000.

Young-Ok Shin is with the Department of Computer Information System, Hanyang Women's College, Seoul, Korea. (phone: +82 2 2290 2380, e-mail: yoshin@swsys2.korea.ac.kr)

Sung-Kong Park and Doo-Kwon Baik are with the Department of Computer Science, Software System Laboratory, Korea University, Seoul, Korea.

Sung-Kong Park (phone: +82 2 925 3706, e-mail: skpark@swsys2.korea.ac.kr)

Doo-Kwon Baik (phone: +82 2 925 3706, e-mail: baik@swsys2.korea.ac.kr)

Keun-Ho Ryu is with the Department of Computer Science, Database Laboratory, Chungbuk National University, Cheongju, Korea. (phone: +82 431 61 2254, e-mail: khryu@dblab.chungbuk.ac.kr)

I. INTRODUCTION

Data warehousing is an application field which deals with continuously time-varying data [1]. It is significant that nearly all warehouses are dependent on relational database technology, yet the relational database model provides little or no real support for temporal data. Objects in the real world are generally perceived to be time-varying. However, relational databases do not capture temporal aspects of the real world. Rather, they only represent a snapshot of real world objects at some instant of time. Therefore, it is difficult to obtain accurate information for time-varying data. Clients of the warehouse may be interested not only in the most up-to-date information, but also in the history of how the source data has evolved. It is therefore important that the warehouse should support temporal queries.

In this paper, first of all, to support temporal data in data warehousing, we combine the temporal layer with the relational database management system which is used as a source database (Fig. 1). Here, we use a layered architecture. A temporal layer (temporal query process system) can obtain accurate information for time-varying data by using the temporal query language. The temporal aggregate function, which is not provided by the existing relational database, provides the various kinds of functions that treat time-varying data. Furthermore, since this is an aggregate function for the time attribute, the history of the event that is occurred in the current world can be processed effectively.

Second, for the way to provide a more accurate temporal materialized view for the time-varying data, the summary table is created using the TATS. The TATS stands for the Temporal Aggregate Tree Strategy. The TATS can be easily used for creating the constant interval set and the time partition by creating the binary tree from the relation that is not ordered in the basis

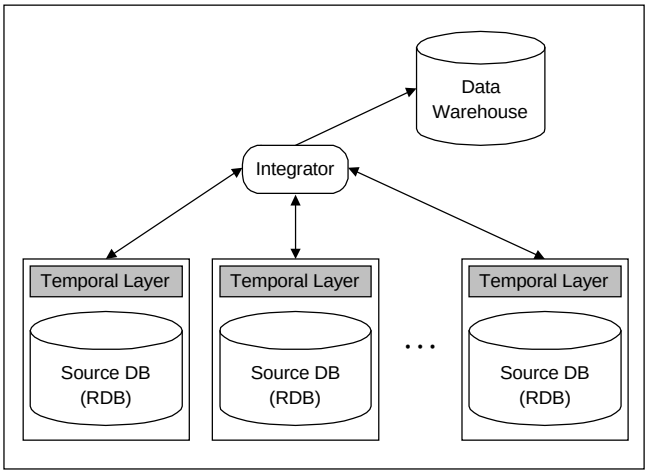


Fig. 1. Architecture of a data warehousing system.

of the time attribute. The TATS is performed in three steps. In the first step, it creates a binary tree by retrieving the tuples that satisfy the condition from the fact table of the data warehouse. In the second step, it creates a temporary relation from the created binary tree. In the third step, it creates a temporal materialized view after aggregating the temporary relation by using the temporal aggregate function.

Finally, the creation method of the materialized view for the existing data warehouse and the data warehouse that uses the suggested TATS have been compared. To compare the performance of the existing data warehouse and the suggested TATS, we evaluate 1) the disk space needed in case of creating the fact table, 2) the number of disk accesses in case of processing the query, and 3) the required size of the memory in case of processing the aggregate function for creating the summary table.

Providing temporal views over non-temporal source data is a very useful feature of a data warehouse. The objective of this paper is to obtain more accurate information for the data, which varies with time by supporting temporal data in a data warehouse. The temporal data warehouse that uses the TATS makes it easy to manage the data that changes by the time by recording the events occurred in the real world clearly. Many applications will benefit from such a temporal data warehouse. For example, a warehouse that stores daily bank account balances can provide audit logs. A company may use a temporal warehouse to keep track of periodic marketing and income figures.

Recently, the interests in the data warehouse have increased, and related research in progress is very active. An important use of data warehousing is to provide temporal views over the history of source data. In a recent paper [2], Yang and Widom are interested in techniques by which the warehouse can materialize relevant temporal views over the history of non-temporal source data. In the paper, views in the data warehouse are constructed from the conceptual temporal base relations using temporal algebra operators. However, it does not mention the aggregation of the temporal data.

Many view maintenance researches have been studied. In [3], it introduces a new algorithm, Eager Compensating Algorithm (ECA), that eliminates the anomalies. [4] and [5] study an efficient evaluation of self-maintainability. In [6], it describes an algorithm called 2 VNL that allows warehouse maintenance transactions to run concurrently with readers. However, these researches are dependent on *only* conventional (relational) databases as a source database. Therefore, it is difficult to obtain accurate information for time-varying data. In this paper, to support temporal data, we combine the temporal query layer with the source database (relational database).

With the growing number of large data warehouses for decision support applications, efficiently executing aggregate queries (queries involving aggregation) are becoming increasingly important [7]. Materialized views involving aggregation are especially important in data warehouses because clients of the warehouse often want to summarize data in order to analyze trends. Quass [8], Chaudhuri and Shim [9] provide maintaining views with aggregation in the general case, where aggregation can include group-by attributes and a view can include several aggregate operators. In [10], it develops powerful query rewrite rules for aggregate queries. Aggregation in these papers, however, can not supply temporal query, especially interval time. Also, the existing data warehouse uses various aggregate functions for the summary, but it is not used for the temporal data. Most of the researches for the temporal aggregate function have been expanded in order to use the aggregate function that has been used in the previous relational database in the temporal model [11]. The authors suggested the TATS which is used for computing the aggregate functions in temporal databases. Although it is as much similar as Kline's work [12], his work has been limited to only a standalone temporal database system. However, we would like to focus this paper on its adaptation of the TATS to temporal data warehouse research. In other words, it is indispensable element to service an efficient computation of time-varying information in a data warehouse. In some researches [13]–[17], Shin introduces a method for summarizing temporal data for time intervals using an aggregate function. It supports time-varying data efficiently and obtains accurate information for time-varying data.

II. TEMPORAL DATA IN DATA WAREHOUSES

1. Temporal Data Model

It is significant that nearly all data warehouses are dependent on relational database technology, yet relational databases provide little or no real support for temporal data. A relational database cannot support for the storage and access of time-varying data. Therefore, it is difficult to obtain an accurate information for time-varying data. The reason is that it is hard to get the effective

Name	Rank	Salary	Start	End
Kim	Associate	30,000	1986	1992
Kim	Full	55,000	1993	now
Lee	Assistant	25,000	1983	1987
Lee	Associate	35,000	1988	1992

Fig. 2. Time-varying data in the real world.

Rank	Salary
Assistant	25,000
Associate	65,000
Full	55,000

(a) The result from a relational database management system

Rank	Salary	Start	End
Assistant	25,000	1983	1987
** Associate	30,000	1986	1987
Associate	65,000	1988	1992
Full	55,000	1993	1998

(b) The result from a temporal query process system

Fig. 3. Comparison of the results from RDBMS and temporal system.

support for the interval time in case of storing the record in the fact table of the data warehouse.

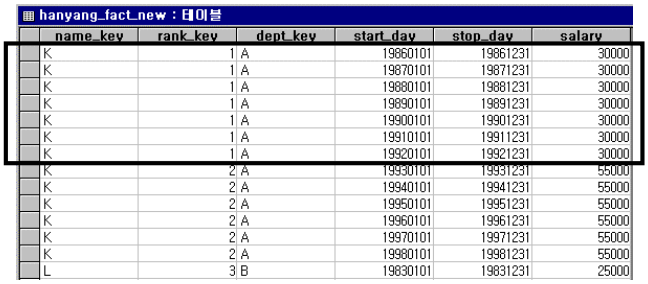
Figure 2 illustrates the example of time-varying data in the real world.

[Query 1] "Compute the sum of salary for each professor by his rank."

Figure 3(a) is the process result of a relational database management system for query 1, while Fig. 3(b) is that of a temporal query process system.

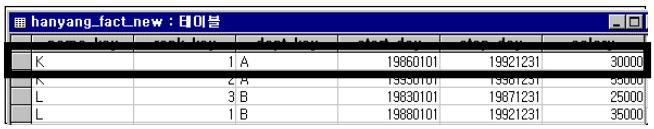
The '**' marks the different results from temporal query, which are identified by interval time. As shown here, a temporal query process system can express time-varying data accurately.

With time-varying data in the real world (Fig. 2), in order to store the record that has the interval time in the fact table of the existing data warehouse, two attributes that are the start time and the end time are required. Furthermore, in order to have the aggregate information, a large number of tuples have to be created for one record. For example, the first tuple 'Kim' has the salary 30,000 from the year 1986 to year 1992; therefore, 7 tuples are needed (Fig. 4). In order to describe the *month* unit instead of *year* unit, much more tuples have to be created. To insert the record that should be described in the interval time into the fact table like this needs a lot of time and space. Furthermore, a lot of problems occur in creating the temporal



name_key	rank_key	dept_key	start_day	stop_day	salary
K	1	A	19860101	19861231	30000
K	1	A	19870101	19871231	30000
K	1	A	19880101	19881231	30000
K	1	A	19890101	19891231	30000
K	1	A	19900101	19901231	30000
K	1	A	19910101	19911231	30000
K	1	A	19920101	19921231	30000
K	2	A	19930101	19931231	55000
K	2	A	19940101	19941231	55000
K	2	A	19950101	19951231	55000
K	2	A	19960101	19961231	55000
K	2	A	19970101	19971231	55000
K	2	A	19980101	19981231	55000
L	3	B	19830101	19831231	25000

Fig. 4. Fact table of the existing data warehouse.



K	1	A	19860101	19921231	30000
L	3	B	19830101	19871231	25000
L	1	B	19880101	19921231	35000

Fig. 5. Fact table of the temporal data warehouse.

materialized view from this fact table. The problem of inserting time of source data, the required space, and the search time is one of the serious problems that must be solved in the existing data warehouse.

In order to solve these problems, this paper does not store the interval time data in the constant interval (year unit), instead, it stores the *interval time of the real world* in the fact table (Fig. 5). Furthermore, the TATS method is introduced for the summarized method for creating the temporal materialized view from the fact table that consists of this time-varying data.

2. Temporal Query Process System

We use a layered architecture as a method to combine a temporal query process system with a source database. The queries written in the temporal language are then converted to SQL queries that are subsequently executed by the underlying source DBMS. No conversion is needed for plain SQL queries. The layered temporal query process system is shown in Fig. 6.

In this architecture, the scanner (temporal syntax analyzer) exports a predicate to read input from standard I/O and to scan text strings. The parser (temporal semantic analyzer) parses a token list. Metadata management controls relational database access in general and access to the metadata of layered architecture in particular.

This architecture implements a temporal query language on top of a relational database management system. The layer converts temporal queries to SQL queries, keeps track of information used by the layer internally, and does some post-processing of the result received from the database management system. More precisely, a transaction with temporal statements is compiled into a single SQL transaction that is executed on the database management system without interference from the layer. The layer simply receives the result and applies some post-processing. There is thus no control module in the layer, and there is minimal

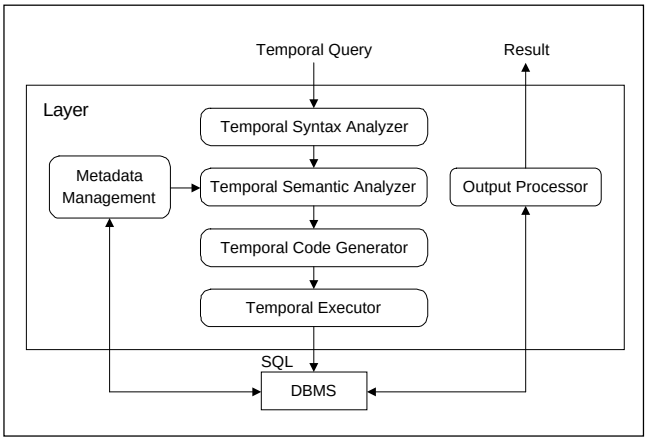


Fig. 6. Temporal query process system.

interaction between the layer and the database management system.

A temporal query, which contains aggregate functions, produces the results about aggregation through temporal query processor.

The processing of a query containing aggregate functions is as follows:

1) Input of temporal query (included aggregate function)
2) Parsing
3) Generation of aggregate query tree
4) Generation of execution tree
5) Compute of aggregation
6) Result

When a query is input, temporal syntax analyzer parses it. If the query contains aggregate functions, it produces an aggregate query tree which expresses the aggregate functions. The temporal code generator optimizes codes and produces the execution tree. The execution tree is a tree that is composed an operation nodes to operate database from the temporal aggregate query. We use an aggregate tree strategy as a computing of aggregation.

III. TEMPORAL DATA AGGREGATION

1. Temporal Aggregate Functions

Unlike the aggregate functions of the existing relational database, the temporal aggregate functions have the characteristic that they are performed not only in the values of object but also in the interval time. The temporal aggregate function, which is not provided by the existing relational database, provides the various kinds of functions that treat time-varying data. The temporal aggregate functions are rising, timefirst, timelast, timemin, timemax, etc. The 'timefirst' function returns the valid time for the oldest tuple in the given relation. The 'timemin' function returns the interval time of the tuple that has the shortest interval time. The 'rising' function returns the interval time for the time when the value of the designated attribute begins to increase. The TATS of the Section IV-2 implements the temporal aggregate functions.

Furthermore, in order to perform the temporal aggregate function, the *time partition* and the *constant interval set* are needed. The 'time partition' is the one that partitions the interval time by the time that the value of the aggregate function changes. The 'constant interval set' signifies the set of the tuples that is included in the same time partition area.

2. The Temporal Aggregate Tree Strategy

The TATS can be easily used for creating the constant interval set and the time partition by creating the binary tree from the relation that is not ordered in the basis of the time attribute. The temporal aggregate tree is the binary tree that satisfies the following conditions:

- The root node stores the start time and the end time for the value of the time attribute. The start time has the value of 0 that is not a negative value, and the last time is the maximum integer value that is defined by the system.
- The root node has the partition attribute value, and the rest of the nodes have the aggregate attribute value.
- The interval time of root node is larger than that of child node.
- The nodes of the same level have the interval time larger than the end time of the left node and smaller than the start time of the right node.

The TATS is performed in three steps. In the first step, it creates a binary tree by retrieving the tuples that satisfy the condition from the fact table. In the second, it creates a temporary relation from the created binary tree. In the third step, it creates a temporal materialized view after aggregating the temporary relation by using the temporal aggregate function. The algorithm in creating this binary tree and the algorithm in creating the temporary relation from the binary tree are as follows:

```
[Algorithm 1] The Binary Tree Creation algorithm
Create_Aggregate_Tree() {
    Do(Read the tuple from the fact table) {
        if (the previous tree does not exist) {
                • Create the new tree and initialize it.
                • Store the tuple value on the node }
            else if (the previous tree exists) {
                    if (the tree with the same partition attribute exists) {
                      • Create a new node in the lower level of the tree
                      • Store the tuple value on the node}
                    else if(the tree with the same partition attribute does
                    not exist) {
                      • Create the new tree and initialize it
                      • Store the tuple value on the node
                      • Link to the root node of the previous tree }
    }while(the tuple exists)
}
```

```
[Algorithm 2] The Temporary Relation Creation algorithm

Create_Temporal_Relation( ) {
    Do {
        if(the left node of the tree exists) {
                • Estimate the 'by' attribute
                • Retrieve the left node
                • DFS_Compute_Tree(temporary_relation, left_ node, 'by'
                attribute)}
        if( the right node of the tree exists) {
                • Estimate the 'by' attribute
                • Retrieve the right node
                • DFS_Compute_Tree(temporal_relation, right_ node, 'by'
                attribute)}
        Move to the next binary tree
    } while(the binary tree exists)
}
```

If the aggregate function is SUM, it returns the total value for the tuple that satisfies the interval time condition from the relation, and it is executed like the following algorithm. The result becomes the temporal materialized view.

```
[Algorithm 3] The Aggregate Function SUM algorithm

while(the tuple exists in the temporary relation) {
    if(the partition attribute of the temporary relation
                = the partition attribute of the inputted tuple ) {
        if(the interval time for the tuple in the temporary relation and
        the interval time for the inputted tuple are same) {
            for(counter value of the temporary relation) {
                return value += aggregate attribute value for the
                tuple in the temporary relation;
                Move to the next aggregate attribute in the same
                tuple;
            }
        }
    }
    Retrieve the next tuple;
}
```

3. An example

This section describes the TATS and the aggregate process of the aggregate function by taking an example. The fact table is illustrated in Fig. 7.

[Query 2] "Compute the sum of salary for each professor (name)."

In [Query 2], the *name* is 'partition attribute', and the *salary* is 'aggregate attribute'.

Name	Rank	Dept	Salary	Start	End
(1) Shin	Assistant	Database	350,000	1987	1990
(2) Shin	Associate	Database	500,000	1991	1993
(3) Shin	Associate	Network	550,000	1991	1995
(4) Lee	Associate	Architecture	600,000	1992	1995

Fig. 7. A faculty relation.

In order to process [Query 2], the TATS is executed in the following steps:

[Step 1] After reading the tuple stored in the given relation one by one, create the binary tree (Fig. 8).
The aggregate tree generated by operating [Query 2] is as follows (Fig. 8). The nodes (a), (b), (d), (e) are valid nodes in the aggregate tree, and each node shows a constant interval. The constant interval of node (c) is the same as that of node (b), so node (c) is linked by node (b). Therefore, 'attribute value to be aggregated' for the interval time from 1991 to 1993 is 1,050,000 by adding 550,000 to 500,000.

[Step 2] Create the temporary relation that stores the attribute value that aggregates based on the partition attribute value from the created aggregate tree by the same interval time (Fig. 9).

[Step 3] Execute the temporal aggregate function SUM from the temporary relation (Fig. 10).

IV. PERFORMANCE EVALUATION

The method that is introduced in this paper reduces the size of the fact table and the disk space in the data warehouse by using the temporal aggregate function. Therefore, to compare the performance of the existing data warehouse and the suggested TATS, we evaluate 1) the disk space needed in case of creating the fact table, 2) the number of disk accesses in case of processing the query, and 3) the required size of the memory in case of processing the aggregate function for creating the summary table.

1. Required Spaces in Case of Creating the Fact Table

In the existing data warehouse, each tuple in the real world must be stored repeatedly in a time interval that users define. So, the number of tuples (T_i) needed to store one tuple of the real world in the fact table is the following:

$$T_i = \frac{Vt_i - Vf_i}{DI}$$ here Vf_i is the start time of the *i*-th tuple, Vt_i is the end time, and DI is the time interval that users define.

If the number of tuples in the real world is N, then the total number of tuples stored in the fact table is

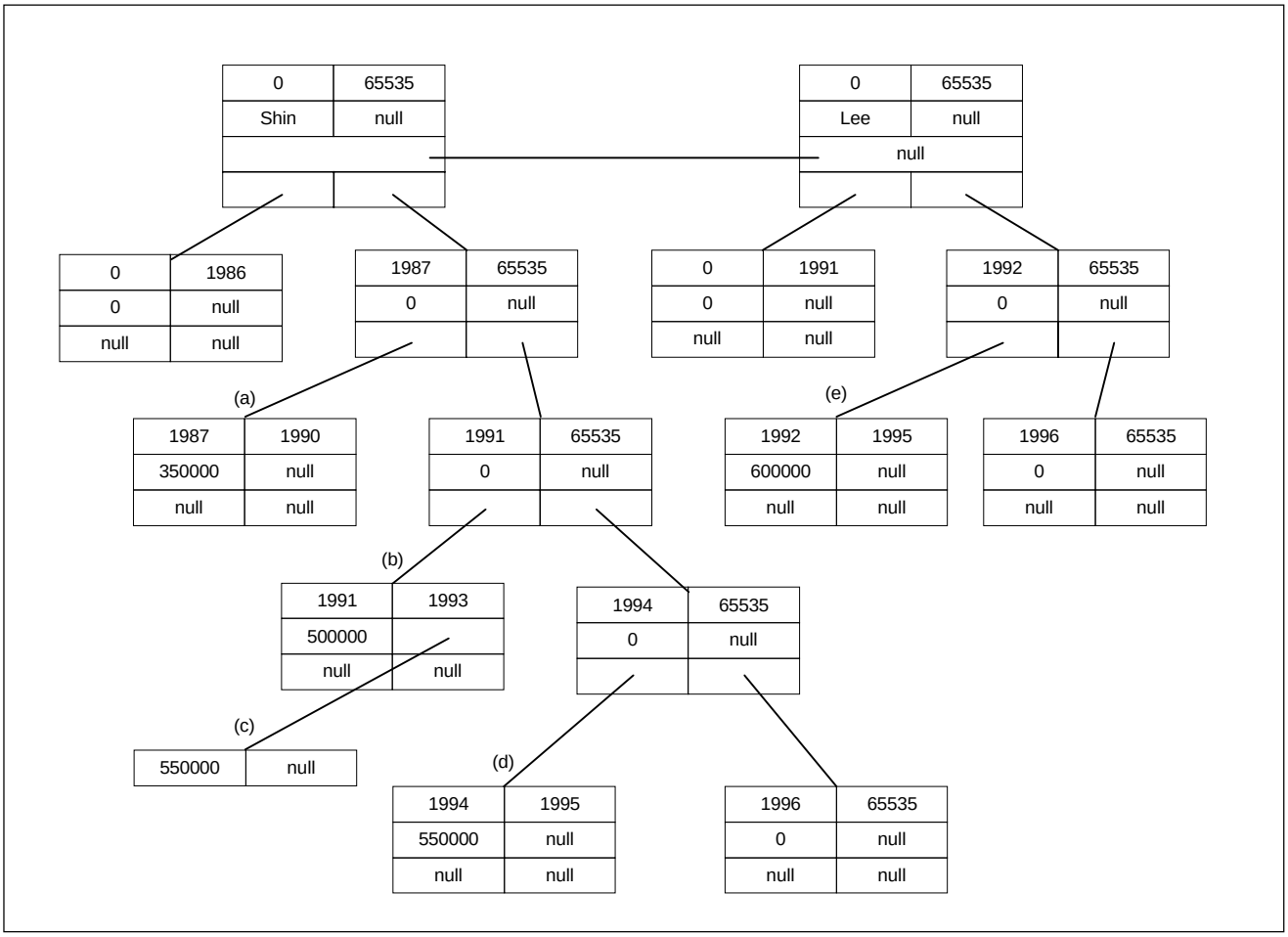


Fig. 8. A generated aggregate tree.

Counter	Partition Attribute	Next Partition attribute	Aggregate Attribute	Next Aggregate Attribute	Start	End	Next Tuple Address
1	Shin	Null	350,000	Null	1987	1990	Oxfffa22d0
2	Shin	Null	500,000	Oxfff8340a	1991	1993	Oxfffa2301
1	Shin	Null	550,000	Null	1994	1995	Oxffa2302
1	Lee	Null	600,000	Null	1992	1995	Null

Oxfff8340a	550,000	Null

Fig. 9. A structure of a temporary relation.

Shin	Total annual salary	Start	End
Shin	350,000	1987	1990
Shin	1,050,000	1991	1993
Shin	550,000	1994	1995
Lee	600,000	1992	1995

Fig. 10. A result of the aggregate function "SUM".

$$T = \sum_{i=0}^{N} T_i = \sum_{i=0}^{N} \frac{Vt_i - Vf_i}{DI} = \overline{V} \times \frac{N}{DI}, \tag{1}$$

$$\overline{V} = \frac{\sum_{i=0}^{N} (Vt_i - Vf_i)}{N} \tag{2}$$

If the TATS is used, the tuple in the real world is mapped into 1 to 1, so the required number of tuples stored in the fact table is

$$T = N \tag{3}$$

2. Number of Disk Accesses in Case of Processing the Query

The relationship for the number of disk accesses between the previous method and the TATS is

$$D_o = \frac{\overline{V}}{DI} \times N = \frac{\overline{V}}{DI} \times D_n . \tag{4}$$

Here, D_o is existing method, D_n is TATS.

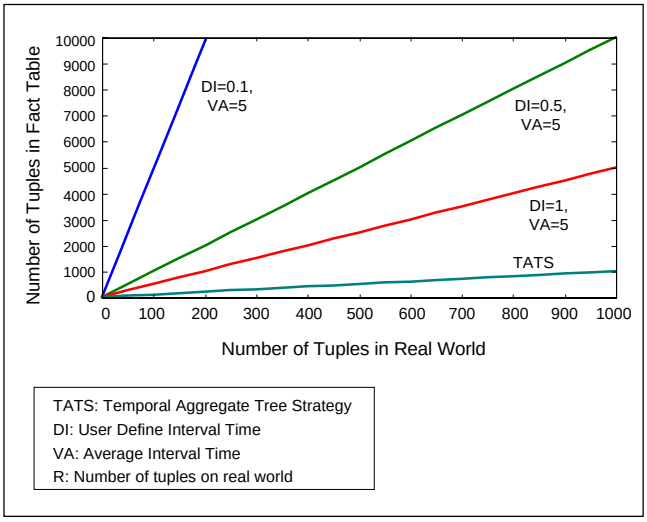


Fig. 11. Number of tuples in the real world and number of tuples in the fact table.

Therefore, the previous method must read $\frac{\overline{V}}{DI}$ times more from the disk. The relationship between the previous method and this method can be illustrated like the following graph (Fig. 11).

3. Required Amount of Memory

This section compares the required amount of memory in order to create the temporal materialized view for the fact table.

A. Required amount of memory in the previous method

The SQL statement to process [Query 2] is as follows:

```
SELECT Name, SUM(Salary), Min(Start), Max(End)
FROM faculty
GROUP BY Name
```

The steps for processing the above query are as follows:

[Step 1] Read the entire table.
[Step 2] Order them by the partition attribute.
[Step 3] Make the temporary table by calculating the aggregate function.

Suppose the number of the partition attribute value is P. The number of tuples in real world for the *i*-th partition attribute is n_i, where $1 \le i \le P$, the start time of the *j*-th tuple for the *i*-th partition attribute is $Vf_{i,j}$ and the end time of it is $Vt_{i,j}$ where $1 \le j \le n_i$. Also, suppose the size of the partition attribute is PAs, the size of the aggregate attribute is AAs, the size of the time attribute is TAs, and the total number of tuples in the entire fact table is T. In addition, if $\overline{n} = \sum_{i=1}^{P} \frac{n_i}{P}$, $\overline{V_i} = \sum_{j=1}^{n_i} \frac{Vt_{i,j} - Vf_{i,j}}{n_i}$, $\overline{V} = \sum_{i=1}^{P} \frac{\overline{V_i}}{P}$, then

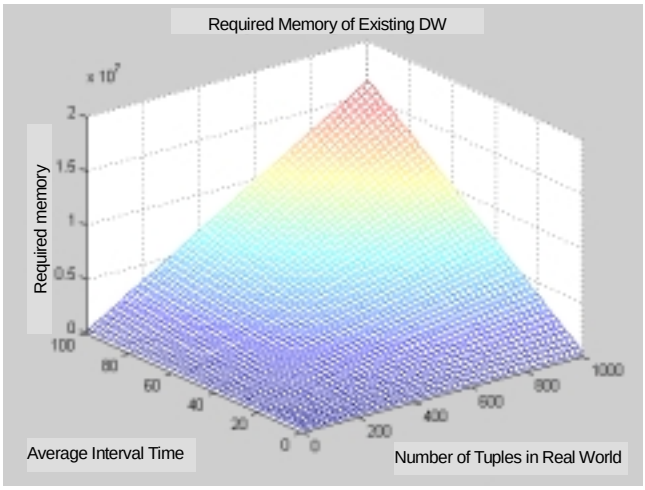


Fig.12. Required memory of existing DW.

Required amount in the 1st step:

$$M_r = T \times (PAs + AAs + TAs) = \sum_{i=1}^{N} \frac{Vt_i - Vf_i}{DI} \times (PAs + AAs + TAs) = \frac{\overline{V}}{DI} \overline{n} P \times (PAs + AAs + TAs) \quad (5)$$

Required amount in the 2nd step:

$$M_s = PAs + AAs + TAs \quad (6)$$

Required amount in the 3rd step:

$$M_c = (PAs + AAs + TAs) \times P \quad (7)$$

Total required amount of memory:

$$M = M_r + M_s + M_c = (PAs + AAs + TAs) \times (1 + P + \frac{\overline{V}\,\overline{n}}{DI} P) \quad (8)$$

In case of [Query 2],

$$PAs = 4, \quad AAs = 4, \quad TAs = 8 \text{ (unit: byte)}$$

$$M_{query} = 16 \times (1 + P + \frac{\overline{V}\,\overline{n}}{DI} P). \quad (9)$$

Figure 12 shows the required memory of existing DW by the size of tuples in real world and average interval time. As shown in Fig. 12, the required memory of an existing DW depends on the amount of the tuples in the fact table.

B. Required amount of memory in the TATS

The required size of memory for processing the aggregate query by using the TATS is the sum of the memories required in each step.

Suppose the size of the root node is Rs, the size of the child node is Cs, the number of child node of the *i*-th tree(the value of the partition attribute) is Cn_i, the size of the connected list of the tree is Ls, the size of tuple of the temporary relation is Tts, the size of the linked list of the temporary relation is Lts, the number of *i*-th linked list is Ln_i, and the number of the constant interval in the *i*-th tree is CI_i.

Aggregate Tree Step: $M_a = \sum_{i=1}^{P}(Rs + Cs \times Cn_i + Ls \times Ln_i)$ (10)

Temporary Relation: $M_t = \sum_{i=1}^{P}\{Tts \times CI_i + Lts \times Ln_i\}$ (11)

Calculation: $M_c = \sum_{i=1}^{P}\{Tts \times CI_i\}$ (12)

Total required amount of memory:

$$M = M_a + M_t + M_c = Rs \cdot P + Cs\sum_{i=1}^{P} Cn_i + (Ls + Lts)\sum_{i=1}^{P} Ln_i + 2Tts\sum_{i=1}^{P} CI_i \quad (13)$$

In case of [Query 2]

$Rs = 28,\ Cs = 24,\ Ls = 8,\ Lts = 12,\ Tts = 32$ (unit: byte),

$$M_{Query} = 28 \cdot P + 24\sum_{i=1}^{P} Cn_i + 20\sum_{i=1}^{P} Ln_i + 64\sum_{i=1}^{P} CI_i \quad (14)$$

In order to insert the new tuple into the tree for the TATS, there are a lot of cases depending on the relationship with the interval time of the previous tuple. The case where the required amount of memory is minimum is the case where the interval time of all the tuples coincide, and in this case the child node and the constant interval time is not added, but only the linked list is added. The case of maximum is the case where the interval time of the new tuple includes the interval time of all previous tuple. In this case, two constant interval time is added, four child nodes are added, and some linked list is added.

<The case of minimum>

$Cn_i = 4$: Four child nodes are needed for describing one interval time.

$Ln_i = n_i - 1$: One linked list is added whenever one tuple is inserted.

$CI_i = 1$: One constant interval time exists for each partition attribute.

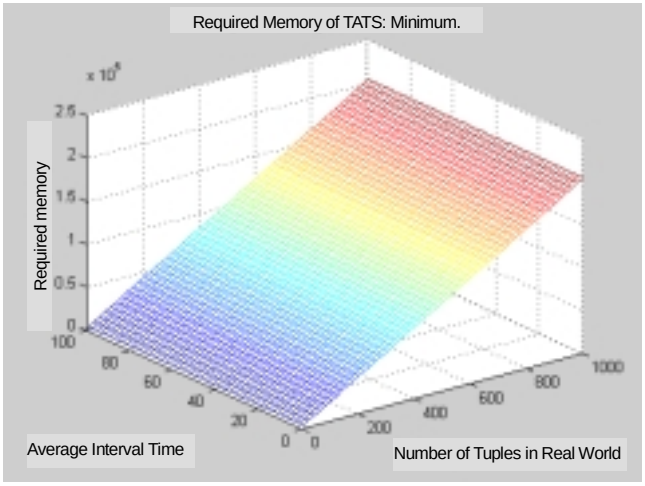


Fig. 13. Required memory of TATS: minimum.

Minimum amount of memory is

$$M_{\min} = 168P + 20\sum_{i=1}^{P} n_i \quad (15)$$

If $\bar{n} = \sum_{i=1}^{P}\frac{n_i}{P}$, then $M_{\min} = 168P + 20\bar{n}P$.

Figure 13 shows the minimum case of the TATS by the amount of the tuples in fact table and average interval time. The required memory of TATS does not depend on average interval time, but it depends on the size of the tuples in the fact table.

<The case of maximum >

$Cn_i = 4n_i$: If one tuple is inserted into the tree, four child nodes are added.

$Ln_i = (n_i - 1)^2$: The number of linked list to be added is the number of constant interval time of previous tree.

$CI_i = 2n_i - 1$: Two constant interval time is added.

Therefore, $M_{\max} = 20\bar{n}^2 P + 184\bar{n}P - 16P$.

Figure 14 shows the maximum case of the TATS by the amount of the tuples in fact table and average interval time.

4. Experimental Result

To evaluate the required memory of TATS in general case, we implement the TATS simulator for aggregate function, and experiment it with sample data.

Sample data is a relation whose attributes are *(partition, start time, end time, aggregate)*. To generate sample data, in TATS simulator, we use uniform deviation and a normal (Gaussian) distribution of specified mean and standard deviation. Uniform deviates are just random numbers that lie within a specified range,

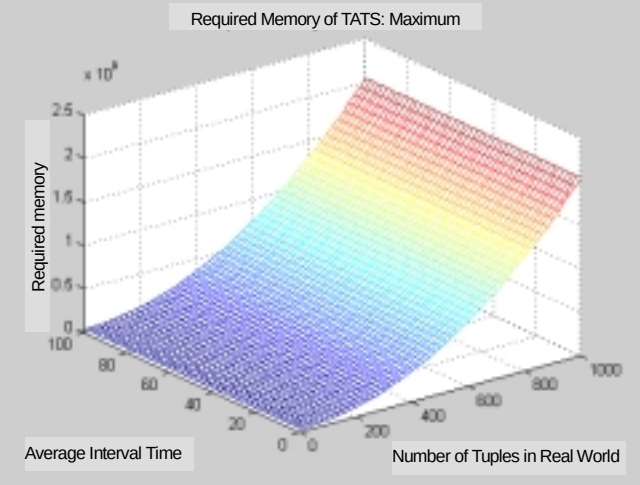


Fig. 14. Required memory of TATS: maximum.

with any one number in the range just as likely as any other. The values of *partition*, *start time*, and *aggregate* are generated by using uniform deviation with specified range. The values of *end time* are generated by adding *start time* and interval time. We generated interval time using normal distribution of specified mean and unit variance.

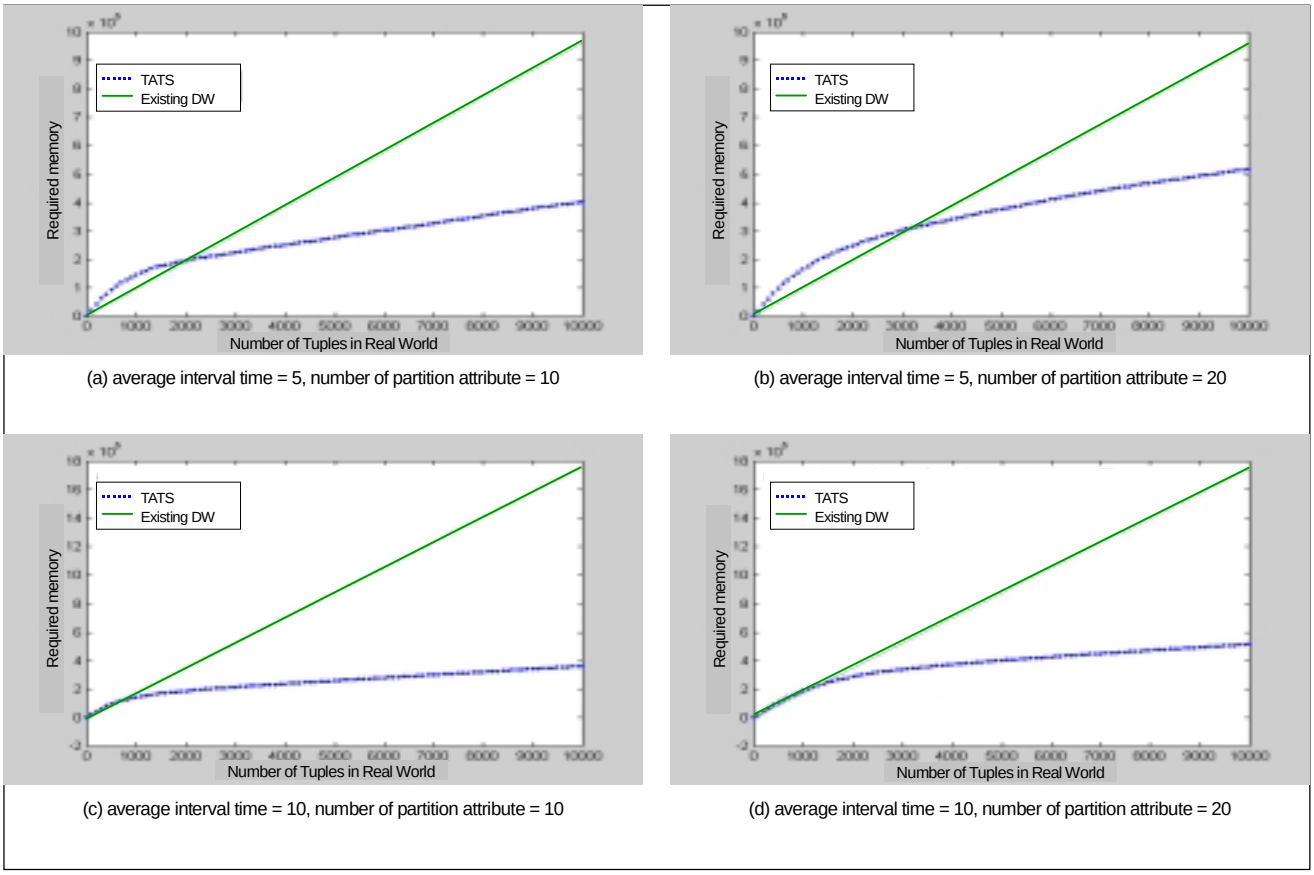
Figure 15 illustrates the comparison between the required amount of memory in the existing data warehouse and the required amount of memory in the TATS according to average interval time, the number of tuples in fact table, and the size of partition attribute value.

As shown in Fig. 15, the angle of inclination of TATS curve is greater than that of existing DW at the beginning, and then the curve slopes gradually up. This status is owing to the probability of splitting the constant interval time is large at the beginning. Splitting constant interval time means add new child nodes to tree. In case where splitting constant interval time never occurs, that is, all constant interval time equals to interval time unit, we don't need to add new child nodes to tree, but just add linked lists to tree. So, the curve slopes gradually up. In case of existing DW, whenever we add a new tuple, we need memory according to the size of interval time of that tuple. So, the curve of existing DW slopes continuously up.

We find that the more tuples exist in fact table and the larger average interval time is, the less memory is required for TATS.

V. CONCLUSIONS

Providing temporal views over non-temporal source data is a very useful feature of a data warehouse. The objective of this



(a) average interval time = 5, number of partition attribute = 10

(b) average interval time = 5, number of partition attribute = 20

(c) average interval time = 10, number of partition attribute = 10

(d) average interval time = 10, number of partition attribute = 20

Fig. 15. Experimental result.

paper is to obtain more accurate information for the data, which varies with time by supporting temporal data in a data warehouse. As suggested in this paper, the temporal data warehouse that uses the temporal aggregation makes it easy to manage the data that changes by the time by recording the events occurred in the real world clearly. Therefore, more accurate information can be provided for a lot of trend analysis. Many applications will benefit from such a temporal data warehouse. For example, a warehouse that stores daily bank account balances can provide audit logs for financial analysts. A company may use a temporal warehouse to keep track of periodic marketing and income figures, personnel transfers, and the history of relevant corporate partnerships.

For that purpose, in this paper we have presented an approach to maintaining temporal data over non-historical information sources in data warehousing. First of all, we combined the temporal query process system into a data warehouse to make more efficient support for temporal data. Also, in this research, the TATS is suggested for the way to aggregate the time-varying data. This method has the advantage that it reduces the space of the fact table and it reduces the number of disk accesses in case of executing the query by comparing with the previous method. Furthermore, the required amount of memory in case of processing the aggregate function for creating the temporal materialized view requires less memory than the existing data warehouse in general. And, we find that the more tuples exist in fact table and the larger average interval time is, the less memory is required for TATS.

Additionally, the TATS that is used to aggregate time-varying data in this paper is an appropriate processing method if the tuples in the relation are not ordered by the time attribute. The problem for the TATS is that the binary tree becomes the skewed tree by the characteristic of time, so it can cause the decrease in the performance since the search time is increased. This problem can cause the delay in the search time and inefficiency in the storage in the large database like the data warehouse. However, these problems can be solved by using the time index suggested by Elmasri [18].

For further research, the research for the aggregate processing method where the tuples can be used in the ordered structure for the time attribute is needed. Furthermore, the purpose of the materialized view is to provide the fast decision support system. Therefore, the research on the view maintenance for processing the query by using the temporal materialized view is needed.

REFERENCES

[1] V. Poe, "Building a Data Warehouse for Decision Support," Prentice Hall, NJ 1996.

[2] J. Yang and J. Widom, "Maintaining Temporal Views Over Non-Temporal Information Sources For Data Warehousing," *Proceedings of the 6th International Conference on Extending Database Technology (EDBT '98)*, Valencia, Spain, March 1998.

[3] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment," Computer Science Dept. Stanford Univ.

[4] D.Quass, A. Gupta, I. Mumick, and J. Widom, "Making Views Self-Maintainable for Data Warehousing," Stanford Univ.

[5] N. Huyn, "Efficient View Self-Maintenance," Dept. of Computer Science, Stanford Univ.

[6] D. Quass, and J. Widom, "On-Line Warehouse View Maintenance for Batch Updates," Stanford Univ.

[7] A. Gupta, Venky Harinarayan, and D. Quass, "Aggregate-Query Processing in Data Warehousing Environments," *Proceedings of the 21st VLDB Conference Zurich*, Switzerland, 1995.

[8] D. Quass, "Maintenance Expressions for Views with Aggregation," Stanford Univ.

[9] S. Chaudhuri and Kyuseok Shim, "Optimizing Queries with Aggregate Views."

[10] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-Query Processing in Data Warehousing Environments," Stanford Univ.

[11] R. Snodgrass, "The Temporal Query Language Tquel," *ACM TODS*, Vol. 12, No. 2, Jun. 1987.

[12] N. Kline, and R. Snodgrass, "Computing Temporal Aggregates," *Proceeding of the Conference ICDE*, 1995.

[13] Y. Shin, D. Baik, K. Ryu, and J. Lee, "Supporting Temporal Data in Data Warehousing," *Journal of Computer Science and Information Management (JCSIM)*, June, 2000.

[14] Y. Shin, D. Baik, and K. Ryu, "Integrating Temporal Data in a Data Warehouse," *Proceedings of the 16th IASTED International Conference*, Feb. 1998.

[15] Y. Shin, D. Baik, and K. Ryu, "Supporting Temporal Data in a Data Warehouse," *Proceedings of the High Performance Computing Conference*, Apr. 1998.

[16] Y. Shin, D. Baik, and K. Ryu, "Summarization Methodology of Temporal Data Warehouse Using Aggregate Tree Strategy," *Proceedings of the ISFST-98*, Oct. 1998.

[17] Y. Shin, D. Baik, and K. Ryu, "Integrating and Managing Temporal Data in a Data Warehouse," *Proceedings of the Computer and their Applications*, Mar. 1998.

[18] R. Elmasri, G. Wuu, and Y. Kim, "The Time Index: An Access Structure for Temporal Data," *Proceedings of the Conference on Very Large Databases*, Aug. 1990.

Young-Ok Shin is an associate professor of Computer Information System Department at Hanyang Women's College, Seoul, Korea and a member of Korea Information Science Society. She received the B.S. and M.S degrees in Computer Science from Soongsil University in 1984 and 1986, respectively. And, she received the Ph.D. degree in Computer Science from Korea University in 2000. With over 15 years, she has studied and implemented various database systems. Her work is focused on data warehousing and temporal data process for maximizing information assets these days. Her research interests are in database management, data warehousing, and temporal database systems. She has published several proceedings and jounals.

Sung-Kong Park is a Ph.D. student of Computer Science & Engineering Department at Korea University, Seoul Korea. He received the B.S. and M.S. degrees in Computer Science from Korea University in 1996 and 1998, respectively. He worked at ETRI as a requested reseach staff for ERP project. His current research areas include Database, Information Integration, Interoperablity, Metadata, and Multi-database System. He has published his articles in several international conferences and jounals.

Doo-Kwon Baik is a full professor of the Computer Science Department, Korea University. He received the B.S. degree in Mathematics from Korea University, Seoul, Korea, in 1974, and the M.S. and Ph.D. degree in Computer Science from Wayne State University at U.S.A., in 1983 and 1986, respectively. His current research areas include Data Engineering, Database, Software Engineering, Modeling and Simulation. He has published over 50 refereed technical articles in various journals, international conferences, and books. He has worked as program committee members for international conferences and workshops. He is a member of the ACM, IEEE-CS, SCS, KISS.

Keun-Ho Ryu is a professor in the Computer Science Department of Chungbuk National University, Cheongju, Korea. He received the B.S. degree in Computer Science from Soongsil University in 1976, and the M.S. and Ph.D. in Computer Science/Engineering from Yonsei University in 1980 and 1988, respectively. He worked at ETRI as well as Korea National Open University. He also joined University of Arizona as a research staff member for TempIS Project. He has published over 70 referred technical articles in various journals, international conferences, and books. His currents research areas include temporal databases, spatiotemporal databases, object and knowledge based system, and knowledge based information retrieval.