Processing Temporal Aggregates in Parallel *

Xinfeng Ye,
Department of Computer Science,
University of Auckland,
New Zealand.

John A. Keane,
Department of Computation,
UMIST,
Manchester, UK.

Abstract

Temporal databases maintain past, present and future data. TSQL2 is a query language designed for temporal databases. In TSQL2, the `GROUP BY` *clause has the temporal grouping property. In temporal grouping, the time line of each attribute value is partitioned into several sections, and aggregate functions are computed for each time partition. This paper describes two approaches to parallelising an algorithm for computing temporal aggregates. The two approaches have been implemented on an SGI PowerChallenge SMP parallel system. The experimental results show that the performance of the two approaches depends on data skew ratio and the number of processors used in the computation.*

1 Introduction

Conventional databases are designed to capture the most recent data, that is, current data. When new data values become available, the existing data values are overwritten by the new values (the old values are removed from the database). Therefore, conventional databases capture a snapshot of reality.

Although conventional databases serve some applications well, they are inadequate for applications in which past or future data are also required. To overcome this inadequacy, temporal databases have been developed [3, 6]. A temporal database supports the storage and querying of information which varies over time. In other words, a temporal database maintains past, present and future data.

In general, a temporal database has a set of time-varying relations. Every time-varying relational schema has two timestamp attributes: *time-start* (T_s) and *time-end* (T_e). The timestamp attributes correspond to the lower and upper bounds of a time interval. In a relation, an attribute value of a tuple is associated with timestamps T_s and T_e if it is continuously valid in the interval $[T_s, T_e]$.

*This work is supported by Auckland University under grant A18/XXXXX/62090/F3414040, and by ESPRIT HPCN Project No 22693.

Aggregate operations are applied to the relations in databases to compute scalar values. Many query benchmarks contain a large percentage of aggregate operations [2, 7]. Therefore, in order to improve the performance of database applications, it is essential to execute aggregate operations efficiently.

In this paper, two parallel algorithms for processing aggregate operations in temporal databases are described. The two algorithms are based on the algorithm in [4]. They have been implemented, and results obtained, on a shared memory multiprocessor (SMP) SGI PowerChallenge.

This paper is organised as follows: in §2, temporal aggregate operations and the algorithm for computing temporal aggregate in [4] are described; §3 introduces the two approaches to parallelising the algorithm on an SMP; the results of the two approaches obtained on an SGI PowerChallenge are discussed in §4; finally, conclusions are given in §5.

2 Temporal Aggregate Operations

Aggregate operations are evaluated over relations to compute a scalar values, e.g. the number of people in the departments of a university, etc. Conventional aggregate operations only show a snapshot of the database. For example, the query in Figure 1(b) will read the tuples in the `Employee` relation and output the number of people in each department at present. Aggregate operations in a conventional database consist of two steps [1]:

1. Set up a tuple to store the result of an aggregate operation;

2. Scan the relation to find the tuples which qualify for the aggregate operation; and, update the aggregate result as appropriate.

TSQL2 [5] is a temporal extension to the SQL-92 query language. TSQL2 extends the `GROUP BY` clause of SQL-92 with temporal grouping. In temporal grouping, the time line of each attribute value is partitioned into several sections (if necessary), and

0-7803-4053-1/97/$10.00 © 1997 IEEE

aggregate operations are computed for each time partition.

For example, assume that an `Employee` relation in a temporal database is shown in Figure 1(a). The `start` and `end` attributes represent the time that a person starts and ends his/her service in a department. The two attributes correspond to *time_start* and *time_end*. ∞ means that a person still works for a department when the query is made. From Figure 1(a), it can be seen that the number of people in a department varies over time. The query in Figure 1(b) counts the number of people in each of the departments during different time periods. Figure 1(c) shows the results of the query. The time line for each department has been partitioned into several sections. A partition point is inserted into the time line of a department if there is a personnel change in the corresponding department at that point of time. Within a time section, the personnel of the corresponding department remains stable.

NAME	DEPARTMENT	START	END
Clark	Physics	2	∞
Tom	Mathematics	1	∞
Peter	Mathematics	3	∞
Bill	Physics	0	5
Phil	Physics	5	∞

(a) Employee Relation

```
SELECT department, COUNT(name)
FROM Employee
GROUP BY department
```

(b) Query on Employee Relation

DEPARTMENT	COUNT	START	END
Mathematics	1	1	2
Mathematics	2	3	∞
Physics	1	0	1
Physics	2	2	4
Physics	3	5	5
Physics	2	6	∞

(c) Query Result

Figure 1

Tuma [8] describes an algorithm for calculating temporal aggregate. The scheme is based on the traditional aggregation technique. However, the tuples in a relation must be scanned twice. The scheme works as follows:

1. The tuples in a relation are scanned to determine the periods of time during which the relation remain unchanged. For example, for the query in Figure 1(b), the tuples in Figure 1(a) will be scanned to partition the time line of each department into several sections. During each time section, the personnel of the corresponding department remains unchanged.
2. The tuples in the relation are scanned again to compute the aggregate value for each of the time periods found at the first scan. For example, for the query in Figure 1(b), at the second scan of the relation in Figure 1(a), the number of people in a department during each of the department's time sections will be counted.

The main drawback of the algorithm is the inefficiency of scanning the tuples in a relation twice.

Kline and Snodgrass [4] describe an algorithm which only requires the tuples in a relation to be scanned once. In the algorithm, trees are used to store the time partitions and the aggregate values of each partition. The trees are formed when the tuples are scanned. Each tree represents the aggregation results of one of the attribute values selected by the `GROUP BY` clause. Each node in the tree represents a time section. When a tuple in a relation is checked, the tree is searched to determine whether the time sections recorded in the tree need to be divided further. If the sections need to be divided further, some nodes representing the new sections are inserted into the tree. The aggregation values recorded in the tree will also be updated when the tuples are checked. After all the tuples have been checked, the leaf nodes in a tree represent the time partitions of the corresponding attribute value.

In this scheme, each node in a tree has five components, *start, end, count, left_child* and *right_child*. *start* and *end* indicate the time period represented by the node. *count* records the number of tuples which are valid within the period. *left_child* and *right_child* are two pointers pointing to the node's children. The two children represent a partition of the time period represented by the node. For example, if a node represents time period $[0, \infty]$, the two children of the node would represent periods $[0, a]$ and $[a + 1, \infty]$ respectively.

For the example in Figure 1, when applying this algorithm, the trees will be constructed as shown in Figure 2. Two attribute values, `Physics` and `Mathematics`, are selected by the `GROUP BY` clause of the query in Figure 1(b). Thus, two trees will store the aggregate results for `Physics` and `Mathematics` respectively.

As shown in Figure 2(a), initially each tree has one node representing time period $[0, \infty]$. Figure 2(b) shows the trees after the first two tuples in Figure 1(a) have been checked. As `Clark`'s tuple is valid

for period $[2, \infty]$, the time line for Physics is partitioned into two sections, [0, 1] and [2, ∞]. Hence, two nodes representing [0, 1] and [2, ∞] are added to the Physics' tree. Due to Clark's tuple, the counter of node $[2, \infty]$ is set to 1. Similarly, due to Tom's tuple, two nodes are added to the Mathematics' tree.

Figure 2(c) shows the trees after the third and the fourth tuple in Figure 1(a) have been checked. As Bill's tuple is valid for $[0, 5]$, time period $[2, \infty]$ is partitioned into two sections, i.e. $[2, 5]$ and $[6, \infty]$. Thus, two nodes $[2, 5]$ and $[6, \infty]$ are inserted into the Physics' tree as $[2, \infty]$'s children. As $[0, 5]$ overlaps with $[0, 1]$, Bill's tuple should also be counted as valid during $[0, 1]$. Thus, the counter of node $[0, 1]$ is incremented. Similarly, as Peter's tuple is valid for period $[3, \infty]$, two nodes, $[1, 2]$ and $[3, \infty]$, are added to the Mathematics' tree.

Figure 2(d) shows the Physics' tree after the last tuple in Figure 1(a) is checked. Phil's tuple is valid for time period $[5, \infty]$. Since $[5, \infty]$ intersects with $[2, 5]$, $[2, 5]$ is partitioned into two sections, [2, 4] and [5, 5]. As a result, two nodes, [2, 4] and [5, 5], are inserted into the Physics' tree. As $[5, \infty]$ overlaps with $[6, \infty]$, the counter for $[6, \infty]$ is incremented.

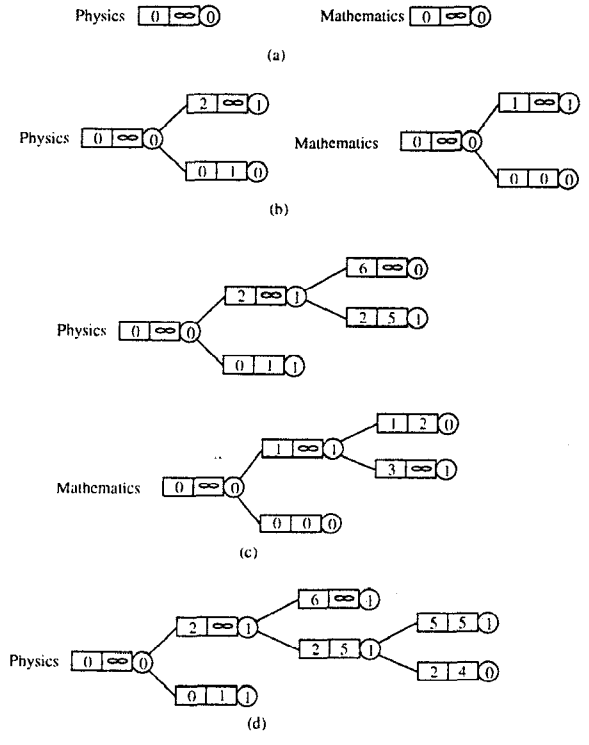

Figure 2

3 Computing Temporal Aggregates in Parallel

In this section, two approaches to parallelising Kline and Snodgrass' algorithm [4] on an SMP parallel system are described.

In the first approach, *group_partition*, the tuples are divided into several groups according to the attribute in the GROUP BY clause of a query. Each processor is responsible for computing the aggregate functions of a group. For example, for the query in Figure 1(b), if there are ten different departments in the Employee relation and five processors, then five groups are formed where each group consists of two departments. Each processor is responsible for computing the aggregate functions of the departments in a group. That is, each processor computes the aggregate functions for two departments. A processor maintains the aggregate trees for the departments in its group. The processor checks all the tuples in a relation and updates the aggregate trees maintained by it accordingly. When a processor has checked all the tuples, the processor outputs the results of the departments in its group.

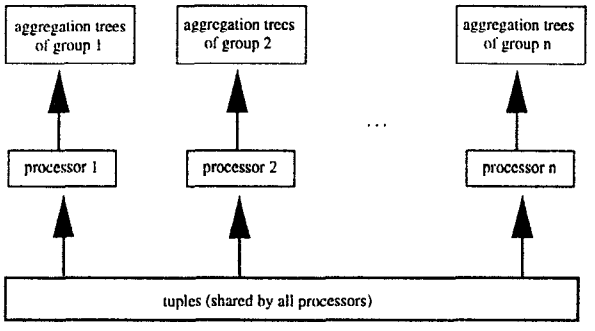

Figure 3

Figure 3 shows the *group_partition* scheme. In the scheme, the tuples of the relations are shared by all the processors. Each processor is given a group of attribute values (e.g. Physics department, Mathematics department etc). The processor is responsible for maintaining the aggregation trees of the group. During the execution, a processor fetches a tuple from the memory. If the tuple corresponds to an attribute value whose aggregation tree is maintained by the processor, the processor updates the aggregation tree according to the algorithm in [4]. Then, the processor fetches the next tuple. If the fetched tuple does not correspond to any of the attribute values maintained by the processor, the processor simply fetches the next tuple from the mem-

ory. These operations are repeated until the processor has checked all the tuples in the memory.

In the second approach, *tree_sharing*, one aggregation tree is created for each of the attribute values selected by the `GROUP BY` clause in a query. For example, for the query in Figure 1(b), an aggregation tree is created for each department. The trees are shared by all processors. In *tree_sharing*, the tuples of a relation are divided into several sections. Each processor only needs to check the tuples in one of the sections. For example, if there are one hundred tuples in a relation and there are five processors, then

(a) the tuples are divided into five sections where each section contains twenty tuples; and

(b) each processor is responsible for checking twenty tuples (i.e. one section).

The aggregation trees are updated by the processors while the tuples are checked. Figure 4 shows the *tree_sharing* scheme.

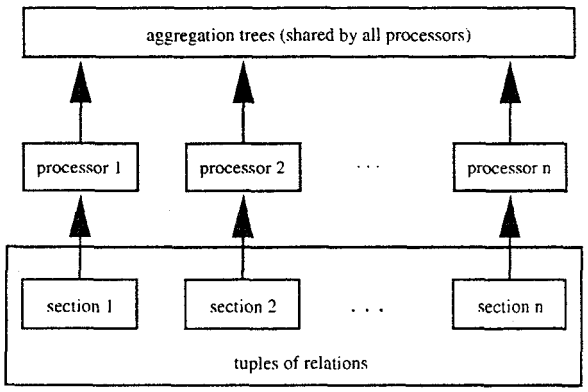

Figure 4

For the *group_partition* scheme, as each processor maintains the aggregation trees of its own group, the operations on the aggregation trees can be carried out on the processors simultaneously. However, each processor has to check all the tuples in a relation. Thus, the *group_partition* scheme trades computation for concurrency.

For the *tree_sharing* scheme, as the aggregation trees are shared by all the processors, if several processors want to update the same aggregation tree, the operations have to be run sequentially. However, instead of checking all the tuples in a relation, each processor only needs to check a small portion of the tuples in a relation. Therefore, the *tree_sharing* scheme is a trade-off between concurrency and computation.

4 Results

The two parallel schemes have been implemented on an SGI PowerChallenge. Each processor is a MIPS R10000 running at 196MHz. The operating system is IRIX 6.2. The programs are compiled using `cc` with `-mp` option.

The size of each tuple in a relation is set to 32 bytes. The relation's lifespan has one million instants, i.e. the values for *time_start* and *time_end* attributes are taken from the range [0, 999999]. The value of the *time_start* attribute of the tuples is generated randomly. The lifespan of a tuple (i.e. the difference between *time_end* and *time_start*) varies between 1 and 1000, and is determined randomly. The number of the distinct attribute values selected by the `GROUP BY` clause has been set to 32[1].

The number of the tuples corresponding to the attribute values selected by the `GROUP BY` clause may vary widely. For example, for the query in Figure 1, the number of people working for the `Mathematics department` may be ten times more than the people working for the `Physics department`. Such an uneven distribution of tuples is called *data skew*.

Assume:

(a) an attribute value, say A, has n tuples associated with it;

(b) an attribute value, say B, has m tuples associated with it;

(c) the number of tuples associated with A is greater than the number of tuples associated with the other attribute values; and

(d) the number of tuples associated with B is less than the number of tuples associated with the other attribute values.

$\frac{n}{m}$ is called the *skew ratio*.

In the *group_partition* scheme, each processor is responsible for maintaining the aggregation trees of some attribute values selected by the `GROUP BY` clause. Data skew may cause some processors to be overloaded. This is because, when the skew ratio is large, the number of tuples in a group, say G, might be significantly higher than the number of tuples in other groups. As a result, the processor responsible for G will perform most of the operations on aggregate trees.

In the *tree_sharing* scheme, the aggregation trees are shared by all processors. Updating an aggregation tree concurrently is not permitted. Therefore,

[1]This means, for the example in Figure 1, there are 32 different departments.

data skew may escalate the memory contention problem. To analyse the effect of data skew several test data sets have been created.

For the first sets of data, the skew ratio is set to 1. That is, the number of tuples associated with each of the 32 attribute values is roughly the same. The tuples associated with different attribute values are stored in a relation in random order.

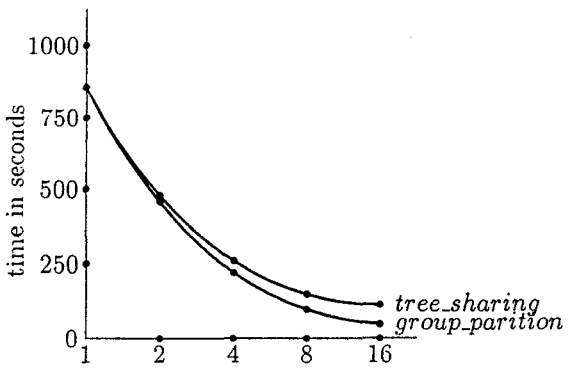

Figure 5 128K Tuples with Skew Ratio 1

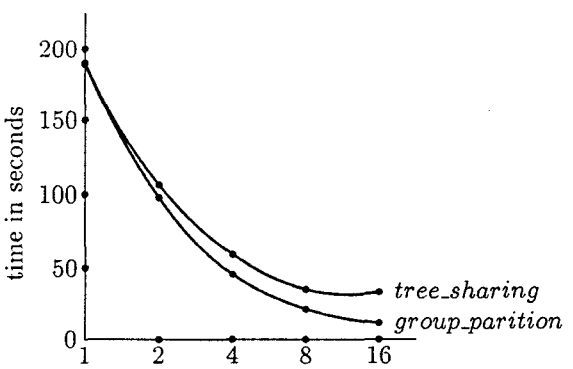

Figure 6 64K Tuples with Skew Ratio 1

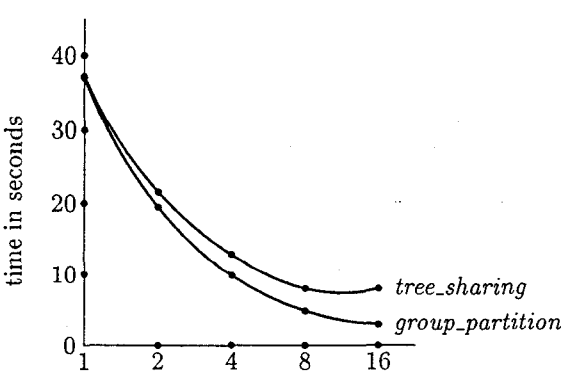

Figure 7 32K Tuples with Skew Ratio 1

The results produced by these sets of data are shown in Figures 5, 6 and 7. The three figures show the running time of the two parallel schemes when the number of tuples in the relation is 128K, 64K and 32K respectively. From the figures, it can be seen that the performance of the two schemes is close to each other when a small number of processors (less than 4) are used. When more processors (more than 4) are used, the performance of the *group_partition* scheme is much better than the *tree_sharing* scheme. In fact, when 16 processors are used, the *group_partition* scheme takes less than half the time required by the *tree_sharing* scheme to complete its operations.

Under these sets of data, the number of tuples in each group is roughly the same. Thus, the load of each processor is also roughly the same. From Figures 5, 6 and 7, it can be seen that, under this circumstance, increasing concurrency is more effective in reducing execution time than decreasing the amount of tuples to be checked by a processor.

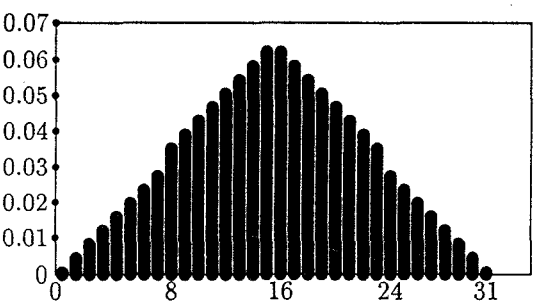

Figure 8 Tuple Distribution when Skew Ratio 128

For the second sets of data, the skew ratio is set to 128. Figure 8 shows the percentage of the tuples associated with the 32 attribute values. In the figure, each attribute value has been given an identifier j ($0 \leq j \leq 31$). The tuples associated with different attribute values are stored in a relation in random order. In the *group_partition* scheme, for a processor i (where $0 \leq i \leq 15$), the identifiers of the attribute values which are in processor i's group are $\{k \mid \frac{32}{n} \times i \leq k \leq \frac{32}{n} \times (i+1) - 1\}$ where n is the number of processors used in a run. For example, if four processors are used in a run, $\{k \mid 0 \leq k \leq 7\}, \{k \mid 8 \leq k \leq 15\}, \{k \mid 16 \leq k \leq 23\}$ and $\{k \mid 24 \leq k \leq 31\}$ are the sets of identifiers in the groups maintained by processors 0, 1, 2 and 3 respectively. Figures 9, 10 and 11 show the results of running the two schemes using these sets of data.

From the figures, it can be seen that, for each test the *group_partition* scheme is better for 2 processors, but shows little improvement from 2 to 4 processors. This is because, according to the distribution of the tuples shown in Figure 8, when four processors are used, processor 1 and 2 are responsible for processing 78% of the tuples in the relation. This means that

most of the operations are carried out on two processors. Thus, the execution time on four processors is only slightly less than with two processors.

From the figures, it can also be seen that the *tree_sharing* scheme outperforms the *group_partition* scheme between 4 and 8 processors. For 16 processors the *group_partition* scheme is better than or equal to the *tree_sharing* scheme.

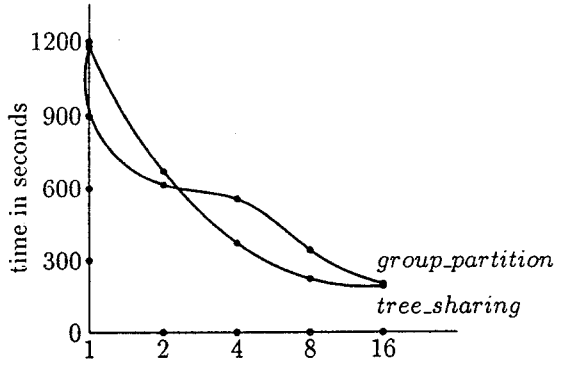

Figure 9 128K Tuples with Skew Ratio 128

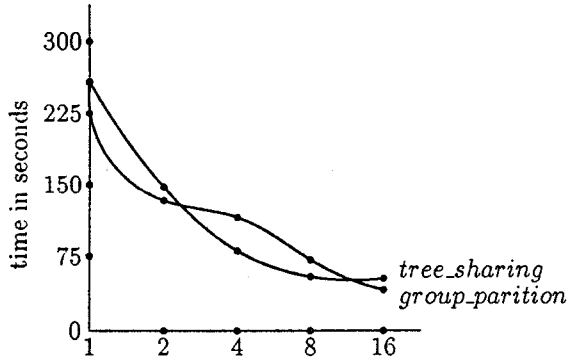

Figure 10 64K Tuples with Skew Ratio 128

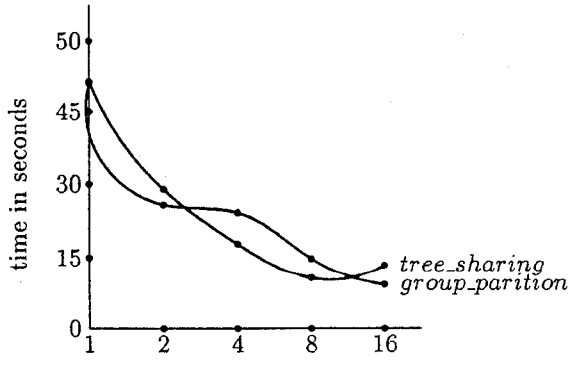

Figure 11 32K Tuples with Skew Ratio 128

5 Conclusions

In this paper, two approaches, *group_partition* and *tree_sharing*, to paralleling an algorithm for computing temporal aggregates [4] have been studied.

In the *group_partition* scheme, the aggregate trees are maintained by different processors. Thus, the operations on the aggregate trees can be carried out simultaneously.

In the *tree_sharing* scheme, aggregate trees are shared by all the processors. Thus, operations on an aggregate tree cannot be carried out by different processors concurrently.

The results show that, when the skew ratio is small and a large number of processors are used (more than 4), the *group_partition* scheme performs better than the *tree_sharing* scheme. However, when the skew ratio is large, the *tree_sharing* scheme outperforms the *group_partition* scheme in most cases. Therefore, in practice, when computing aggregate functions in parallel, a more efficient scheme can be chosen based on skew ratio and the number of processors used in execution.

References

[1] R. Epstein, Techniques for Processing of Aggregates in Relational Database Systems, UCB/ERL M7918, Computer Science Department, University of California at Berkeley, 1979

[2] J. Gray, *The benchmark handbook for database and transaction processing systems*, Morgan Kaufmann, 1991

[3] N. Kline, An update of the temporal databases bibliography, *ACM SIGMOD Record*, **22**(4), pp. 66-80, 1993

[4] N. Kline and R. Snodgrass, Computing temporal aggregates, *Proceedings of 11th International Conference on Data Engineering*, pp. 222-231, IEEE, 1995

[5] R. Snodgrass, I Ahn, G, Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Hensen, W. Kafer, N. Kline, K. Kulkanri, T.Y. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada, TSQL2 language specification, *ACM SIGMOD Record*, **23**(1), pp. 65-86, 1994

[6] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, 1993

[7] TPC, *TPC benchmark^{TM}*, Transaction processing performance council, 1994

[8] P.A. Tuma, Implementing historical aggregates in TempIS, Master Thesis, Wayne State University, 1992