# Processing Temporal Aggregates over Networked Workstations

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

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

**Abstract**

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 parallel algorithms, *data_partition* and *group_partition*, which compute temporal aggregates over a network of workstations.

In the *group_partition* scheme, each workstation maintains the entire aggregate tree for some attribute values selected by the `GROUP BY` clause. Thus, some workstations may be overloaded while others are idle for most of the time. In the *data_partition* scheme, all the workstations participate in constructing the aggregate trees in the first phase of the scheme. Thus, the load is evenly distributed across the workstations in the system in the first phase of the scheme. However, before the second phase starts, workstations must exchange the aggregate trees generated at the first phase.

A simulator has been used to test the performance of the two algorithms. The results show that the performance of algorithm *group_partition* is slightly better than *data_partition*.

## 1 Introduction

A temporal database supports the storage and querying of information which vary over time [3, 7]. 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 interval $[T_s, T_e]$.

Aggregate operations are applied to the database relations to compute a scalar value. Aggregate operations are very important in database applications: many query benchmarks contain a large percentage of aggregate operations [2, 8]. Therefore, in order to achieve good performance of database applications, it is necessary to execute the aggregate operations efficiently.

*This work is supported by Auckland University under grant A18/XXXXX/62090/F3414079, and by the UK EPSRC under grant GR/J48979.

In this paper, two algorithms for processing aggregate operations in temporal databases over a network of workstations are described. The two algorithms are based on the algorithm in [4]. A simulator has been written to simulate the execution of the two algorithms over a network of workstations. The experimental results obtained from the simulator will also be discussed.

## 2 Temporal Aggregate Operations

Aggregate operations are evaluated over relations to compute a scalar value, e.g. the number of people in the departments of a university, etc. Conventional aggregate operations [1] 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*.

TSQL2 [6] 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 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 service in a department. The two attributes correspond to *time_start* and *time_end*. $\infty$ 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 periods of time. Figure 1(c) shows the results of the query. In the result, 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 that department at that point of time. Within a time section, the personnel of the corresponding department remains stable.

| name | department | start | end |
|---|---|---|---|
| John | Statistics | 2 | $\infty$ |
| David | Chemistry | 1 | $\infty$ |
| Bruce | Chemistry | 3 | $\infty$ |
| Bob | Statistics | 0 | 5 |
| Gary | Statistics | 5 | $\infty$ |

(a)

0-7803-4778-1/98 $10.00 © 1998 IEEE

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

| department | count | start | end |
|---|---|---|---|
| Chemistry | 1 | 1 | 2 |
| Chemistry | 2 | 3 | $\infty$ |
| Statistics | 1 | 0 | 1 |
| Statistics | 2 | 2 | 4 |
| Statistics | 3 | 5 | 5 |
| Statistics | 2 | 6 | $\infty$ |

(c)

Figure 1

In [4] an algorithm for calculating temporal aggregate is described in which 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 [4]'s 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 pointers 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 [4]'s algorithm, the trees will be constructed as shown in Figure 2. Two attribute values, Statistics and Chemistry, are selected by the `GROUP BY` clause of the query. Thus, two trees will be set up to store the aggregate result for Statistics and Chemistry 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 John's tuple is valid for period $[2, \infty]$, the time line for Statistics is partitioned into two sections, [0, 1] and [2, $\infty$]. Hence, two nodes representing $[0, 1]$ and $[2, \infty]$ are added to Statistics' tree. Due to John's record, the counter of node $[2, \infty]$ is set to 1. Similarly, due to David's tuple, two nodes are added to Chemistry's tree.

Figure 2(c) shows the trees after the third and the fourth tuple in Figure 1(a) have been checked. As Bob's record 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 Statistics' tree as $[2, \infty]$'s children. As $[0, 5]$ overlaps with $[0, 1]$, it means Bob's record should also be counted as valid during $[0, 1]$. Thus, the counter of node $[0, 1]$ is incremented. Similarly, as Bruce's tuple is valid for period $[3, \infty]$, two nodes, $[1, 2]$ and $[3, \infty]$, are added to Chemistry's tree.

Figure 2(d) shows Statistics' tree after the last tuple in Figure 1(a) is checked. Gary's tuple is valid for time period $[5, \infty]$. As $[5, \infty]$ 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 Statistics' tree. As $[5, \infty]$ overlaps with $[6, \infty]$, the counter associated with $[6, \infty]$ is incremented.

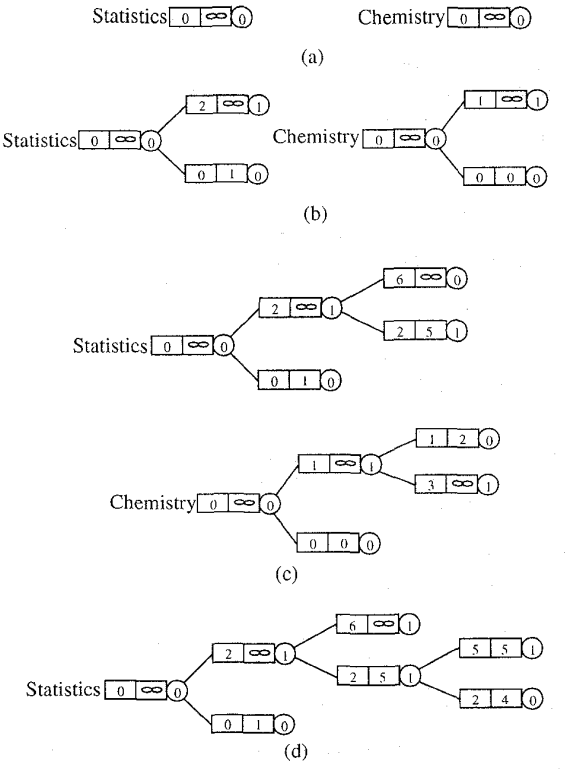


Figure 2

## 3 Compute Temporal Aggregates in Parallel

### 3.1 Basic Principles

In this section, two approaches to parallelise [4]'s algorithm over networked workstations are described. It is assumed that (a) a relation has been partitioned into $n$ sections where $n$ is the number of workstations participating in computing the aggregate functions, and (b) the tuples in a section are stored on one workstation.

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 workstation 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 workstations, then five groups are formed where each group consists of two departments. Each workstation is responsible for computing the aggregate functions of the departments in a group. That is, each workstation computes the aggregate functions for two departments. A workstation maintains the aggregate trees for the departments in its group. First a workstation checks the tuples stored on it. For the tuples belonging to the departments for which the workstation is responsible for computing the aggregate functions, the workstation updates the aggregate trees maintained by it according to [4]'s algorithm. For the other tuples, the workstation sends them to the appropriate workstations which are responsible for processing the tuples. Once a workstation completes checking the tuples stored on it, it starts processing the tuples sent to it by the other workstations. When all the received tuples have been processed, the workstation outputs the results stored in the aggregate trees.

The second algorithm *data_partition* consists of two phases. During the first phase, each workstation processes the tuples stored on it and constructs an aggregate tree for each attribute value in the GROUP BY clause whose tuples are stored on the workstation. For example, in Figure 1(b), a workstation creates an aggregate tree for each of the departments during the first phase. As in the first algorithm, the attribute values in the GROUP BY clause are partitioned into several groups. In the second phase, each workstation is responsible for producing the final results for the attribute values in one group. During the second phase, a workstation (i) sends the aggregate trees of the attribute values whose final results will be produced by other workstations to the appropriate workstations, and (ii) merges the aggregate trees received from the other workstations with the corresponding ones held by the workstation. For example, in Figure 1(b), if there are ten departments and five workstations, each workstation will be responsible for producing the final results of two departments. In the second phase, each workstation (i) sends out the aggregate trees of the eight departments whose final results are not produced by the workstation, and (ii) receives two aggregate trees from each of the other workstations. Once all the received trees have been merged, the workstation outputs the results stored in the trees.

## 3.2 The Algorithms

First algorithm *group_partition* is described. It is assumed that the algorithm is applied to workstation $i$.

1. **for** all the tuples stored in local memory
2. **if** a tuple records the attribute value whose aggregate tree is kept by workstation $i$
3. **then** update the relevant aggregate tree
4. **else** /* i.e. the aggregate tree of the attribute value recorded in the tuple is not kept by workstation $i$ */
5. pack the tuple into a message and send the message to the workstation which keeps the aggregate tree of the attribute value recorded in the tuple

   **endif**

   **endfor**
6. process the tuples sent by the other workstations

In line 5, when sending a tuple to another workstation, there is no need to send all the items recorded in the tuple. It is only necessary to send the information needed for constructing the aggregate tree, i.e. the attribute value, and the *start* and *end* time. This approach reduces the amount of information exchanged amongst the workstations.

Next the algorithm *data_partition* is described. It is assumed that the algorithm is applied to workstation $i$.

7. **for** each tuple stored in the local memory
8. update the relevant aggregate tree
9. **endfor**
10. pack the aggregate trees of the attribute values for which workstation $i$ is not responsible for producing the final result into messages; and, send the messages to appropriate workstations
11. **for** each of the aggregate trees received from other workstations
12. merge the received trees with the corresponding aggregate trees maintained by workstation $i$

    **endfor**

## 3.3 Data Skew

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 Chemistry department maybe ten times more than the people working for the Statistics department. This 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 *skew ratio*.

In the *group_partition* scheme, each workstation is responsible for maintaining the aggregate trees of some attribute values selected by the GROUP BY clause. Data skew might cause some of the workstations being 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 workstation which is responsible for $G$ will perform most of the operations on aggregate trees.

In the *data_partition* algorithm, at the first phase all

the workstations participate in constructing the aggregate trees for all the attribute values. Therefore, at the first phase, the load is shared amongst all the workstations in the system. Although in the second phase of the algorithm, the workstations which handle the aggregate trees of the large tuple groups might still be overloaded, it remains to be seen whether the sharing of load in the first phase makes the *data_partition* algorithm more efficient than the *group_partition* algorithm.

## 4 Empirical Results

A simulator has been written to simulate the execution of the two algorithms over a network of workstations. The simulator uses the message passing libraries in PVM. The simulator runs on a 275MHz Alpha 2100 4/275 with 512M bytes RAM. Another Alpha 2100 is set up to measure the communication time when messages are exchanged between workstations. A dummy program is run on this machine. The program receives and unpacks the messages sent by the simulator. For each received message, the program sends a reply to the simulator. The two machines are connected through a lightly loaded Ethernet. In the simulator, when two workstations exchange a message, the message is first sent to the Alpha where the dummy program resides. When a reply to the message is received from the dummy program, the simulator delivers the message to the destination workstation. The time for exchanging a message is set as half of the elapsed time between sending a message to and receiving a reply from the dummy program.

When algorithm *group_partition* is implemented, instead of packing each tuple into a message (line 5 of the algorithm), several tuples which are sent to the same workstation are packed into the same message. This will reduce the communication set up and packing cost. In the simulator, up to 1000 tuples are packed into a message.

The tuples associated with different attribute values are stored in a relation in random order. The relation has been divided into $n$ sections which contain the same number of tuples where $n$ is the number of workstations used in the computation. A section is stored on one of the workstations in the system.

The size of each tuple in a relation is set to 128 bytes. The relation has a lifespan of 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].

Cache hit rate is one of the many factors that affects the speed up of a parallel system. For some applications, partitioning the data set will increase the cache hit rate. As a result, it is possible to have *super linear* speed up when applications are parallelised. That is, when running the applications on $n$ machines, compared with running the applications on one machine, the speed up is greater than $n$ [5]. Super linear speed up was observed in several cases during the experiment. In order to verify the results, the algorithms were run on a single Alpha 2100 4/275 for one data set containing 128K tuples with 32 attribute values and skew ratio 1 and another data set containing 4K tuples with 1 attribute value[2]. The execution time for the 128K and the 4K tuple sets is 1244.407 seconds and 25.023 seconds respectively. It can be seen that, although the size of the 128K tuple set is 32 times of the 4K tuple set, the execution time for the 128K tuple set is about 49 times of the execution time of the 4K tuple set. This indicates that the cache hit rate does play an important role in the performance of the algorithm. As shown later, in the experiment, due to communication overhead, the speed up is less than 32 when 32 workstations are use in the computation.

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 results produced by these sets of data are shown in Figures 3, 4 and 5. The three figures show the running time of the two parallel schemes when the number of tuples in the relation is 32K, 64K and 128K respectively. From the figures, it can be seen that the performance of the two schemes is very close. As explained earlier, when running on a single workstation, super linear speed up has been observed. However, the scale of speed up shown in Figures 3 to 5 is less than that achieved when running the algorithms on a single workstation. This is because the cost of exchanging the tuples and aggregate trees reduces the gains obtained in improvement in the cache hit rate.

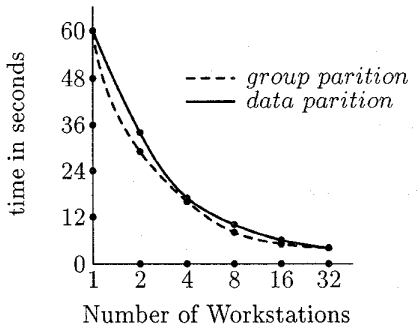


Figure 3 Time Comparison for 32K Tuples with Skew Ratio 1

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

[2]This is because the simulator simulates a system containing up to 32 workstations. With 32 workstations for 32 attribute values, each workstation is responsible for constructing the aggregate tree of one attribute value. Thus, for a 128K tuple set with skew ratio equal to 1, there are 4K tuples associated with each attribute value. Hence, each workstation will handle 4K tuples.

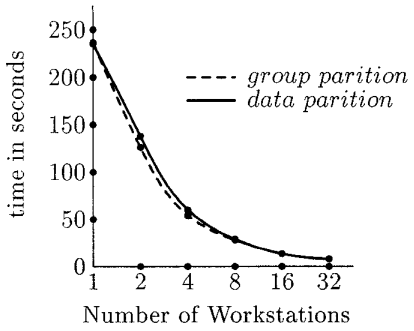

Figure 4 Time Comparison for 64K Tuples with Skew Ratio 1

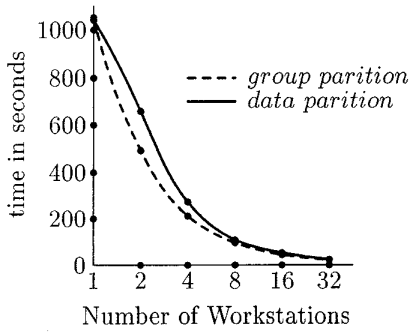

Figure 5 Time Comparison for 128K Tuples with Skew Ratio 1

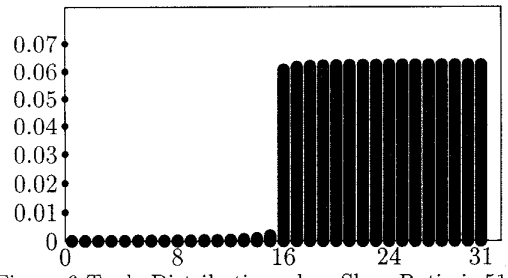

Figure 6 Tuple Distribution when Skew Ratio is 512

For the second sets of data, the skew ratio is set to 512. Figure 6 shows the percentage of the tuples associated with the 32 attribute values. In the figure, each attribute value has been given an identifier $j$ ($0 \leq j \leq 31$). For a workstation $i$ (where $0 \leq i \leq 15$), the identifiers of the attribute values for which workstation $i$ is responsible for processing the aggregate trees are $\{k \mid \frac{32}{n} \times i \leq k \leq \frac{32}{n} \times (i+1) - 1\}$ where $n$ is the number of workstations used in computation. For example, if four workstations are used, $\{k \mid 0 \leq k \leq 7\}, \{k \mid 8 \leq k \leq 15\}, \{k \mid 16 \leq k \leq 23\}$ and $\{k \mid 24 \leq k \leq 31\}$ are the sets of identifiers in the groups maintained by workstation 0, 1, 2 and 3 respectively. Figures 7, 8 and 9 show the results of running the two schemes using these sets of data.

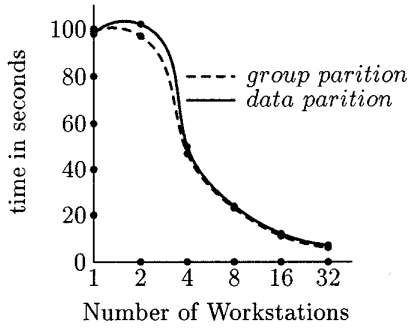

Figure 7 Time Comparison for 32K Tuples with Skew Ratio 512

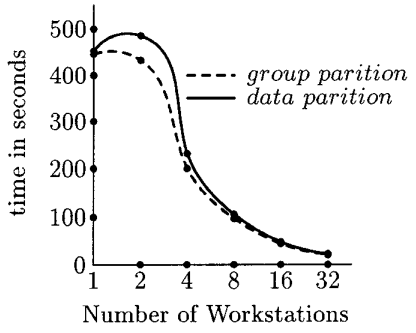

Figure 8 Time Comparison for 64K Tuples with Skew Ratio 512

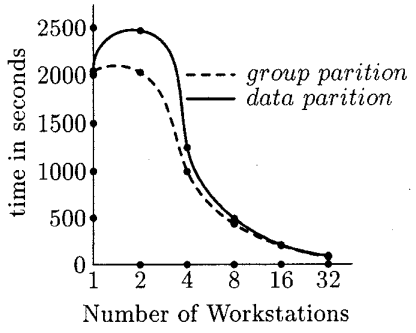

Figure 9 Time Comparison for 128K Tuples with Skew Ratio 512

From the figures, it can be seen that, when two workstations are used, (a) the execution time of algorithm *group_partition* is similar to executing the algorithm on a single machine; (b) the time to run algorithm *data_partition* is slightly longer than running the algo-

rithm on a single workstation. This is because, when two workstations are used, according to Figure 6 one of the workstations is responsible for producing the aggregate trees for 99.3% of the tuples. For algorithm *data_partition*, the aggregate trees generated in the first phase will be sent to the appropriate workstations for merging before the second phase starts. Merging two trees is the same as inserting the leaf nodes of one tree into another tree. As the time intervals of most of the tuples intersect with each other, the number of leafs in the aggregate trees is roughly the same as the number of tuples. Hence, the number of insertion carried out on the aggregate trees are similar for the cases where one and two workstations are used. Therefore, when running on two workstations, the cost in exchanging the aggregate trees makes the *data_partition* algorithm perform worse than running on a single workstation.

From the figures, it can be seen that, apart from the case where two workstations are used, the performance of the two algorithms are similar to each other. As explained above, this is because the number of leaf nodes in the aggregate trees is almost the same as the number of tuples. Thus, (a) the amount of information being exchanged amongst the workstations in algorithm *data_partition* is almost the same as the amount of information exchanged in algorithm *group_partition*; and (b) the amount of insertion operations carried out on the aggregate trees in algorithm *data_partition* is almost the same as the one in algorithm *group_partition*. Point (b) means that, although the load of the workstations at the first phase of algorithm *data_partition* is balanced, the information processed by the workstations in the second phase of the algorithm will overload the workstations to the same extent as with algorithm *group_partition*. Thus, the *data_partition* scheme does not perform better than the *group_partition* scheme. However, this is due to the fact that the time intervals of the tuples are generated randomly. Thus, the time intervals of the tuples hardly coincide with each other. As a result, the number of leaf nodes in the aggregate trees is similar to the number of tuples. For some applications, the time intervals of many tuples might coincide with each other, e.g. many students enter and graduate from the universities at the same time. In this case the number of leaf nodes in the aggregate trees will be much less than the number of tuples. As a consequence, the number of insertion operation carried out in the second phase of algorithm *data_partition* will be much lower than the insertion operations carried out in *group_partition*. Thus, in this kind of applications, it is possible that *data_partition* outperforms *group_partition*.

## 5 Conclusions

In this paper, two approaches, *group_partition* and *data_partition*, to parallelise [4]'s algorithm for computing temporal aggregates over a network of workstations have been studied. In the *group_partition* scheme, each workstation maintains the aggregate trees of some attribute values selected by the `GROUP BY` clause. Thus, when the skew ratio is high, some of the workstations are overloaded while the others are idle for most of the time. In the *data_partition* scheme, all the workstations participate in constructing the aggregate trees in the first phase of the scheme. Thus, the load is evenly distributed across the workstations in the system at the first phase of the scheme. However, before the second phase starts, workstations must exchange the aggregate trees generated at the first phase. The results show that the *group_partition* scheme performs slightly better than the *data_partition* scheme for all the data tested. This means that, for the *data_partition* scheme, the cost of exchanging the aggregate trees outweighs the performance gains obtained through having a balanced load at the first phase of the execution of the scheme.

As the test data are generated over a life span of 1 million instances, the start and end time of the tuples hardly coincide with each other. As a result, many time intervals are generated. Thus, the size of the aggregate trees is very large. Hence, the costs of exchanging the aggregation trees over the network and merging the aggregate trees is relatively high compared with exchanging tuples over the network. Future work will test how the data generated over a shorter life span affect performance of the two schemes.

## 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, Vol. 22, No. 4, pp66-80, 1993

[4] N. Kline and R. Snodgrass, Computing temporal aggregates, Proceedings of 11th International Conference on Data Engineering, pp222-231, IEEE, 1995

[5] H.S. Morse, Practical Parallel Computing, AP Professional, 1994

[6] 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, Vol. 23, No. 1, pp65-86, 1994

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

[8] TPC, TPC benchmark$^{TM}$, Transaction processing performance council, 1994