Distributed and Parallel Databases, 16, 123–163, 2004


# Main Memory-Based Algorithms for Efficient Parallel Aggregation for Temporal Databases

DENGFENG GAO
JOSE ALVIN G. GENDRANO
BONGKI MOON
RICHARD T. SNODGRASS rts@cs.arizona.edu
MINSEOK PARK
BRUCE C. HUANG
JIM M. RODRIGUE
*Computer Science Department, University of Arizona, Tuscon, AZ 85721-0077, USA*

**Recommended by:** Calton Pu

**Abstract.** The ability to model the temporal dimension is essential to many applications. Furthermore, the rate of increase in database size and stringency of response time requirements has out-paced advancements in processor and mass storage technology, leading to the need for parallel temporal database management systems. In this paper, we introduce a variety of parallel temporal aggregation algorithms for the shared-nothing architecture; these algorithms are based on the sequential Aggregation Tree algorithm. We are particularly interested in developing parallel algorithms that can maximally exploit available memory to quickly compute large-scale temporal aggregates without intermediate disk writes and reads. Via an empirical study, we found that the number of processing nodes, the partitioning of the data, the placement of results, and the degree of data reduction effected by the aggregation impacted the performance of the algorithms. For distributed result placement, we discovered that Greedy Time Division Merge was the obvious choice. For centralized results and high data reduction, Pairwise Merge was preferred for a large number of processing nodes; for low data reduction, it only performed well up to 32 nodes. This led us to a centralized variant of Greedy Time Division Merge which was best for the remaining cases. We present a cost model that closely predicts the running time of Greedy Time Division Merge.

**Keywords:** aggregation tree algorithm, cost model, data reduction, parallel temporal aggregation, result placement, temporal declustering

## 1. Introduction

An *aggregate* is a query language construct that computes a single value over a set of tuples. SQL includes five aggregates: `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`. *Aggregate functions* partition the input relation according to the value of a column; SQL accomplishes this through a `GROUP BY` clause.

Aggregate functions are an essential component of data query languages and are heavily used in many applications such as data warehousing. Several prominent query benchmarks contain aggregate operations [18]; all but one of the 17 TPC-D benchmark queries involve aggregates [16]. Hence, efficient execution of aggregate functions is an important goal.

*Table 1*. Sample database and sample temporal aggregations.

| (a) Data tuples | | | | (b) Result of count | | | (c) Result of max salary | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Salary | Start | Stop | Count | Start | Stop | Max | Start | Stop |
| Richard | 40 K | 18 | $\infty$ | 1 | 7 | 8 | 35 K | 7 | 8 |
| Karen | 45 K | 8 | 20 | 2 | 8 | 12 | 45 K | 8 | 20 |
| Nathan | 35 K | 7 | 12 | 1 | 12 | 18 | 40 K | 20 | $\infty$ |
| Nathan | 37 K | 18 | 21 | 3 | 18 | 20 | | | |
| | | | | 2 | 20 | 21 | | | |
| | | | | 1 | 21 | $\infty$ | | | |

Unfortunately, aggregate computation is traditionally expensive, especially in a temporal database where the problem is complicated by having to compute the intervals of time for which the aggregate value holds. Consider the sample table in Table 1(a), listing the salaries of employees and when these salaries are valid, indicated by closed-open intervals. Finding the (time-varying) number of employees (Table 1(b)) involves computing the temporal extent of each value, which requires determining the tuples that overlap each temporal instant. Similarly, finding the time-varying maximum salary (Table 1(c)) involves computing the temporal extent of each resulting value.

In this paper, we present several new parallel algorithms for the computation of temporal aggregates on the shared-nothing architecture [15], which is the best known and increasingly prevalent strategy to build a scalable computing platform using off-the-shelf processors.

We start with the (sequential) Aggregation Tree algorithm [10] and propose several approaches to parallelize it. The performance of the parallel algorithms relative to various data set and operational characteristics is our main interest. Given the high scalability of the shared-nothing architecture and rapidly growing memory capacity of computing platforms, our interest is in developing parallel algorithms that can maximally exploit available resources to quickly compute temporal aggregates that are infeasible when intermediate disk writes and reads are required, as in other algorithms.

This paper is organized as follows. Section 2 gives a review of related work; the following section presents the sequential algorithm on which we base our parallel algorithms. Our proposed algorithms to compute parallel temporal aggregates are then described in Section 4. Sections 5 and 6 present empirical results obtained from the experiments performed on a shared-nothing Pentium cluster. We modify two of the algorithms to exploit time-partitioning in Section 7. Section 8 summarizes the experiments. The cost model of Greedy Time Division Merge is given in Section 9. Finally, Section 10 concludes the paper.

## 2. Related work

Simple algorithms for evaluating scalar aggregates and aggregate functions were introduced by Epstein [5]. A different approach employing program transformation methods

to systematically generate efficient iterative programs for aggregate queries has also been suggested [6]. Snodgrass et al. extended Epstein's algorithms to handle temporal aggregates [14]; these were further extended by Tuma [17] and by Kline [9, 10]. The resulting algorithms are quite effective in a uniprocessor environment. However, because they are inherently sequential, they all suffer from poor scale-up performance, indicating the need to develop parallel algorithms for computing temporal aggregates.

Early research on developing parallel algorithms focused on the framework of general-purpose multi-processor machines. Bitton et al. proposed two parallel algorithms for processing conventional aggregate functions [1]. The Subqueries with a Parallel Merge algorithm computes partial aggregates on each partition and combines the partial results in a parallel merge stage to obtain a final result. Another algorithm, Project By_list, exploits the ability of the parallel system architecture to broadcast tuples to multiple processors. The Gamma database machine project [3] implemented similar scalar aggregates and aggregate functions on a shared-nothing architecture. A few parallel algorithms for handling temporal aggregates were presented [22], but for a shared-memory architecture. Hence, these algorithms have inherently limited scalability.

Recently, Moon et al. [11] proposed a disk-based algorithm that can compute temporal aggregates for a database larger than the available memory of a stand-alone or shared-nothing platform. Yang and Widom [20, 21] proposed an index structure that can be used to compute temporal aggregates incrementally for an entire database or for a temporal range. Zhang et al. [23] was concerned with minimizing I/O in a sequential architecture, by using a new index structure to implement the more general range-temporal aggregation problem, in which the temporal aggregate is restricted by a time interval and a key range. The parallel algorithms proposed in this paper aim at exploiting all available memory of a shared-nothing parallel computing platform to quickly compute large-scale temporal aggregates without intermediate disk accesses in a scalable manner.

## 3. The aggregation tree

The parallel temporal aggregation algorithms proposed in this paper are based on the (sequential) Aggregation Tree algorithm (SEQ) designed by Kline [10]. An *aggregation tree* is a binary tree that tracks the number of tuples whose timestamp periods contain an indicated time span. Each node of the tree contains a start time, an stop time, and a count.

The tree has several global properties. The leaf nodes partition the time line, ordered sequentially with no holes. (As before, the periods are closed-open, so equivalently the stop time of one node must be identical to the start time of the next leaf node.) The period of a branch node is the union of the periods of its child nodes (hence, the nodes at any level partition the time line). The single root node contains all of time. When an aggregation tree is initialized, it begins with a single node containing $\langle 0, \infty, 0 \rangle$ (see the initial tree in figure 1).

In the example from the previous section, four tuples from the argument relation (Table 1(a)) are inserted into an empty aggregation tree. If either the start or stop time is not somewhere in the aggregation tree, a node split occurs. The start time value, 18, of the first entry to be inserted splits the initial tree, resulting in the updated aggregation tree

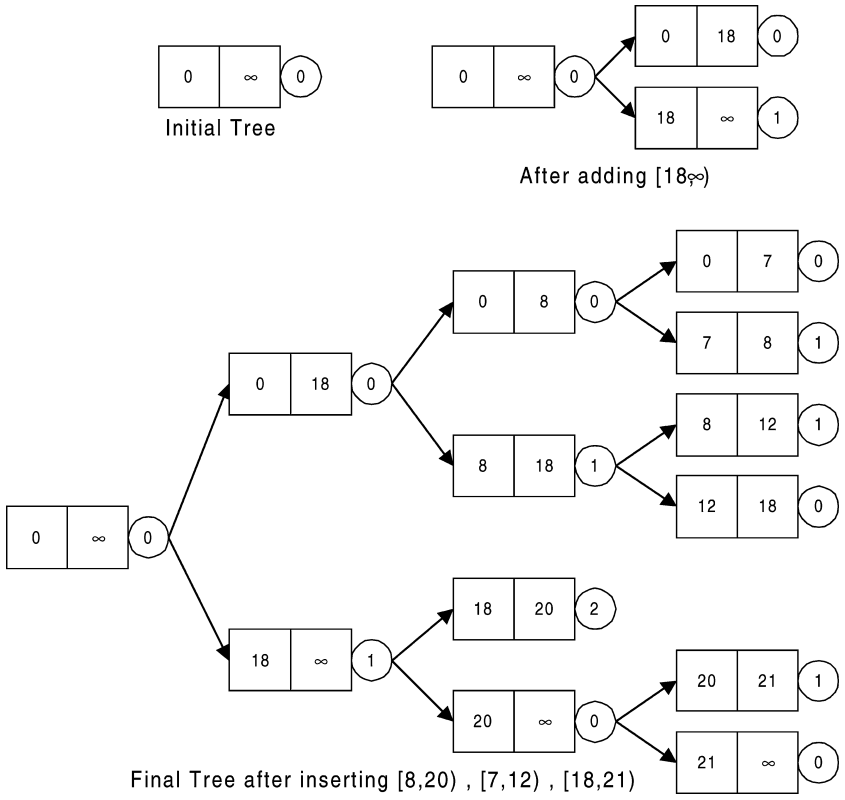



*Figure 1.* Example run of the sequential (SEQ) aggregation tree algorithm, for count.

shown in figure 1. Because the original node and the new node share the same stop date of $\infty$, a count of 1 is assigned to the new leaf node $\langle 18, \infty, 1\rangle$. The aggregation tree after inserting the rest of the records in Table 1(a) is shown at the bottom of figure 1. A period spanning several leaf nodes can update several branch nodes. The algorithm updates only as deep in the tree as needed. For example, inserting the period [8, 20) would update the counts for the [8, 18) and [18, 20] branch nodes. As there are six unique timestamps in the example argument relation, there are six leaf nodes.

It should be noted that the order of tuple insertion into the aggregation tree affects its performance, though not its aggregated result. If the tuples are sorted via the start time and inserted in that order, the aggregation tree would look more like a linked list, causing insertions to be slower than insertions into a balanced binary tree. The aggregation tree is more balanced if the relation is randomly ordered by time.

Once this tree is created, we can enumerate the result through a depth-first traversal, maintaining the intermediate counts along the way (see figure 2). To compute the number of tuples for the period [8, 12) in this example, we simply take the count from the leaf node [8, 12) (which is 1), and add its parents' count values. Starting from the root, the sum of the

```
outputresult(node n, int count):
  if branch node
    return outputresult(n.left, count+n.count) ∪
                outputresult(n.right, count+n.count)
  else return <n.begintime, n.endtime, count+n.count>
```

*Figure 2*. Computing the result given an aggregation tree.

parents' counts is $0 + 0 + 1 = 1$ and adding the leaf count, gives a total of 2. The six leaf nodes of the aggregation tree correspond to the six tuples in the result of the aggregate (see Table 1(b)). Aggregates other than `Count` can be computed by storing other information in each node of the aggregation tree.

The aggregation tree is useful for computing aggregates over a single time dimension, such as *valid time* or *transaction time* [13]. (*Bitemporal aggregates* over two-dimensional time domains [8] are much less prominent, and are not considered here.) *Correlated aggregate functions* such as finding the average salary per department (say via a `GROUP BY` clause in SQL) can be done by creating a separate aggregation tree for each group-by value or by creating a modified aggregation tree containing the group-by values, as discussed in detail by Kline [10]. The result is the aggregate value along with the group-by values(s), which can then be equi-joined with the original input relations if needed. These extensions can all be applied to the algorithms we present below to support general aggregate processing.

Though SEQ correctly computes temporal aggregates, it is still a sequential algorithm, bounded by the resources of a single processor machine. The entire aggregation tree must be constructed before the result may be computed. This makes a parallel method for computing temporal aggregates desirable.

## 4. Parallel processing of temporal aggregates

In this section, we propose five parallel algorithms for the computation of temporal aggregates. We start with two simple parallel extensions to the SEQ algorithm, the Single Aggregation Tree (abbreviated SAT) and Single Merge (SM) algorithms. We then go on to introduce the Pairwise Merge (PM) and Time Division Merge with Centralization (TDM + C) algorithms, which both require more coordination but are expected to scale better. After that, we present the Time Division Merge (TDM) algorithm, a variant of TDM + C, which distributes the resulting relation across the processors, as differentiated from the centralized results produced by the other algorithms.

### *4.1. Single aggregation tree (SAT)*

The first algorithm, SAT, extends the Aggregation Tree algorithm by parallelizing disk I/O and the initial projection operator. For Count, all that are needed are the start and stop times

from the underlying tuples. For the other aggregates, the value being aggregated over from the tuple is also needed. Each worker node reads its data partition in parallel, constructs the valid-time periods for each tuple, sends these periods (along with the column being aggregated, if relevant) to the coordinator. The central coordinator receives the periods from all the worker nodes, builds the complete aggregation tree, and returns the final result to the client.

We note in passing that SM (as well as the algorithms to follow) can be easily generalized to evaluate range-temporal aggregates [23], in which the temporal aggregate is restricted by a time interval and a key range, both of which can be accomplished by the worker nodes in the first part of SAT.

While SAT should have slightly better performance than the sequential version, it is clear at the outset that this algorithm doesn't scale. We mention this algorithm only to provide a starting point for the other algorithms, which are all based on SAT.

### *4.2. Single merge (SM)*

The second parallel algorithm, SM, is more complex than SAT, in that it includes computational parallelism along with I/O parallelism. Each worker node builds a local aggregation tree, in parallel. The worker node then traverses its aggregation tree in DFS order, propagating the count values to the leaf nodes. The leaf nodes now contain the full local count for the periods they represent, and the branch nodes are discarded. (At least half of the nodes will be branch nodes.) The worker nodes sends its leaf nodes to the coordinator.

Unlike the SAT coordinator, which inserts periods into an aggregation tree, the SM coordinator merges each of the leaves it receives using a variant of merge-sort that operates on periods (no aggregation tree is constructed at the coordinator), as indicated in figure 3. The use of this efficient merging algorithm is possible because the worker nodes send their leaves in a temporally sorted order. Finally, after all the worker nodes finish sending their leaves, the coordinator returns the final result to the client.

### *4.3. Pairwise merge (PM)*

The third parallel algorithm, Pairwise Merge (see figure 4), attempts to obtain better performance by replacing the global synchronization step with lg $p$ localized synchronization steps. Which two worker nodes are paired in each localized synchronization step is determined dynamically by the query coordinator.

| |
|---|
| *Step* 1. Client request |
| *Step* 2. Build local aggregation trees |
| *Step* 3. Send leaf nodes to coordinator, which merges them into one list |
| *Step* 4. When all worker nodes have completed, return result to client |

*Figure 3.* Major steps for the single merge algorithm.

| |
|---|
| *Step* 1. Client request |
| *Step* 2. Build local aggregation trees |
| *Step* 3. While not final aggregation tree merge between 2 nodes |
| *Step* 4. Return result to client |

*Figure 4*. Major steps for the pairwise merge algorithm.

A worker node is available for merging when its local aggregation tree has been built. The worker node informs the query coordinator that it has completed its aggregation tree. The query coordinator then arbitrarily picks another worker node that had previously completed its aggregation tree, thereby allowing the two worker nodes to merge their leaves, again, using merge-sort. Then, the query coordinator instructs the worker node with the least number of leaf nodes to send the leaves to the other node, the "buddy worker node", which does the merging of leaves.

Once a worker node finishes transmitting leaves to its buddy worker node, it is no longer a participant in the query. This buddying-up continues until the query coordinator ascertains that only one worker node is left, which contains the completed aggregation tree. The query coordinator then directs the sole remaining worker node to submit the results directly to the client. Figure 5 provides a conceptual picture of this "buddy" system.

A portion of a PM aggregation tree may be merged multiple times with other aggregation trees. The merge algorithm is a merge-sort variant operating on two sorted lists as input (the local list and the received list). This merge is linear in the number of leaf nodes to be merged.

### *4.4. Time division merge with centralization (TDM + C)*

Like PM, the fourth parallel algorithm, TDM + C, also extends the aggregation tree method by employing both computational and I/O parallelism (see figure 6). The main steps for this algorithm are outlined in figure 7.

***4.4.1. Overall algorithm.*** TDM + C starts when the coordinator receives a temporal aggregate request from a client. Each worker node builds a local aggregation tree and propagates the interior counts to the leaf nodes. The worker nodes then exchange minimum (earliest) start time and maximum (latest) stop time values to ascertain the overall timeline of the query.

The leaves of a local aggregation tree are evenly split into $p$ local partitions, consisting of a period and a tuple count. Because each partition is split to have the same (or nearly the same) number of tuples, local partitions can have different durations. The local partition set (containing $p$ partitions) from each processing node is then sent to the coordinator.

The coordinator takes all $p$ local partition sets (a total of $p^2$ local partitions are created by $p$ worker nodes) and computes a global partition set of the time line, as $p$ partitions (how this is done is discussed in the next section). Effectively, the local partition sets are equi-depth histograms [12] approximating the distribution of tuples across the time line;

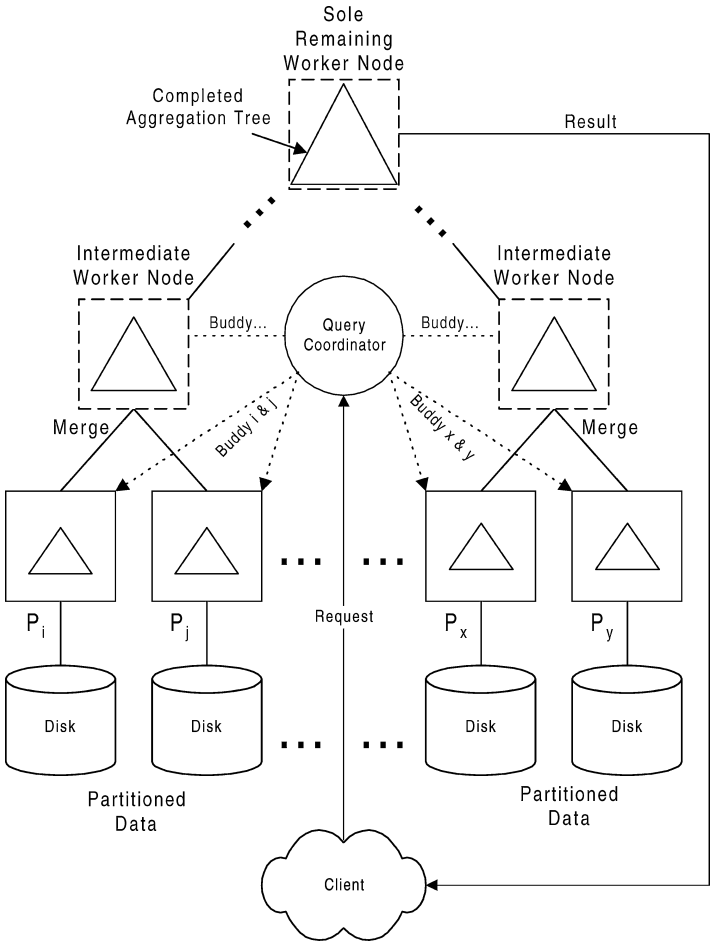



*Figure 5*. Pairwise merge (PM) algorithm.

the global partition set is a global equi-depth histogram across the entire data set. The local partition set is of size $p$ to enable it to scale as the number of processors increases.

After computing the global time partition set, the coordinator then naively assigns the period of the $i$th partition to the $i$th worker node, and broadcasts the global partition set to all the nodes. The worker nodes then use this information to decide which local aggregation tree leaves to send, and to which worker nodes to send them. Note that periods that span more than one global partition period are split and each part is assigned accordingly (split periods do not affect the correctness of the result).

Each worker node merges the leaves it receives with the leaves it already has to compute the temporal aggregate for its assigned global partition. When all the worker nodes finish merging, the coordinator polls them for their results in sequential order. The coordinator concatenates the results and sends the final result to the client.

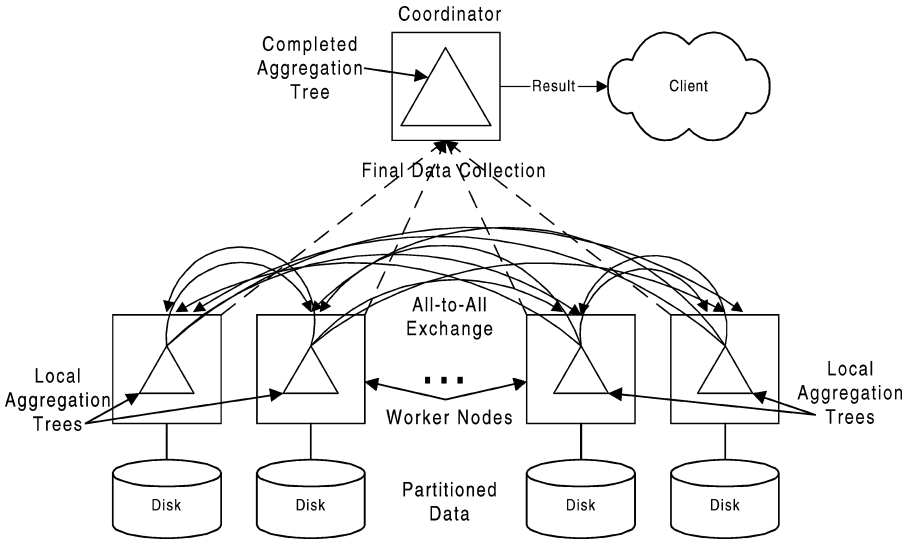



*Figure 6.* Time division merge with centralizing step (TDM + C) algorithm.

| |
|---|
| *Step* 1. Client request |
| *Step* 2. Build local aggregation trees |
| *Step* 3. Calculate local partition sets |
| *Step* 4. Calculate global partition set |
| *Step* 5. Exchange data and merge locally |
| *Step* 6. Globally merge results |
| *Step* 7. Return result to client |

*Figure 7.* Major steps for the TDM + C algorithm.

***4.4.2. Calculating the global partition set.*** Recall that the coordinator receives from each worker node a local partition set, consisting of $p$ contiguous partitions; each partition is associated with a tuple count. The goal is to temporally distribute the computation of the final result, with each node processing roughly the same number of leaf nodes to evenly distribute the second phase of the computation. (We asume a *homogeneous* architecture, containing multiple identical processors.)

As an example, figure 9 presents 9 local partitions from 3 worker nodes. The number between each hash mark segmenting a local timeline represents the number of leaf nodes within that local partition. The total number of leaf nodes from the 3 worker nodes is $50 \cdot 3 + 15 \cdot 3 + 30 \cdot 3 = 285$. The best plan allocates $\frac{285}{3} = 95$ leaf nodes to each worker node. Figure 8 illustrates the computation of the global partition set.

We modified the SEQ algorithm to compute the global partition set, using the local partition information sent by the worker nodes. We treat the worker node local partition

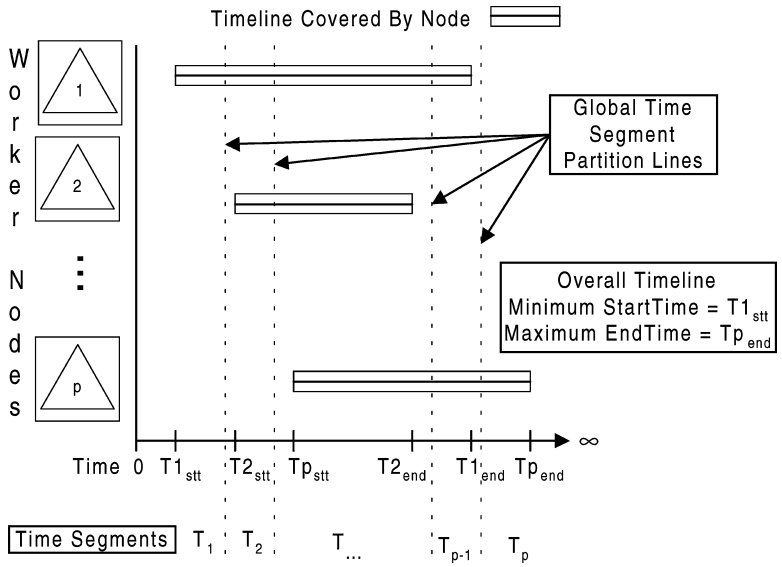



*Figure 8.* Timeline divided into $p$ partitions, forming a global partition set.

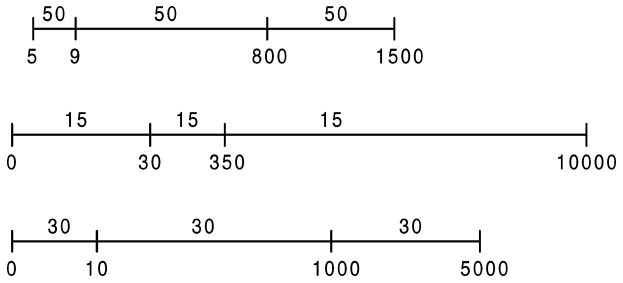
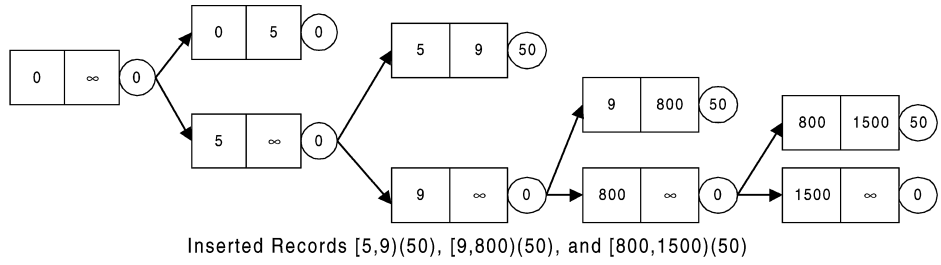


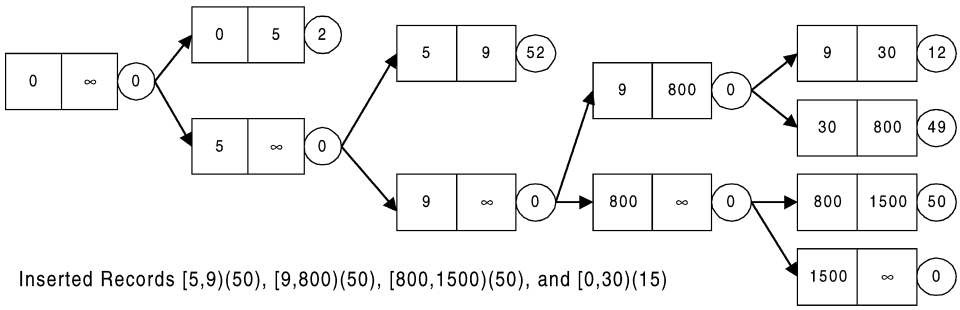

*Figure 9.* Local partition sets from three worker nodes.

sets as periods, inserting them into the modified aggregation tree. From figure 9, the first period to be inserted is [5,9)(50), the fourth is [0,30)(15), the seventh is [0,10)(30), and the ninth(last) is [1000,5000)(30). This use of the aggregation tree is entirely separate from the use of this same structure in computing the aggregate. Here we are concerned only with determining a division of the timeline into $p$ contiguous periods, each with approximately the same number of leaves.

There are three main differences between our Modified Aggregation Tree algorithm used in this portion of TDM + C and the original Aggregation Tree [10] used in step 2 of figure 7. First, the "count" field of this aggregation tree node is incremented by the count value of the local partition being inserted, rather than by 1. Second, every branch node must have a count value of 0. When a leaf node is split and becomes a branch node, its count is split proportionally between the two new leaf nodes based on the durations of their respective




(a) First three local partitions




(b) After partition 4 is added

*Figure 10.* Intermediate aggregation tree.

time periods. The count of this new branch becomes 0. Third, during an insertion traversal for a record, if the search traversal diverges to both subtrees, the record count is split proportionally between the 2 sub-trees.

As an example, suppose we inserted the first three local partitions, and now we are inserting the fourth one, [0,30)(15). The current modified aggregation tree before inserting the fourth local partition is shown in figure 10(a). Notice that for leaf node [5,9)(50), the count value is set to 50 instead of 1 (first difference).

The second and third differences are exemplified when this fourth local partition is added. At the root node, we see that the period for this fourth partition overlaps the periods of the left sub-tree and the right sub-tree. In the original aggregation tree, we simply added 1 to a node's count in the left sub-tree and the right sub-tree at the appropriate places. Here, we see the third difference. We split this partition count of 30 in proportion to the durations of the left and right sub-trees. The root left sub-tree contains a period [0,5) for a duration of 5 time units. The fourth local partition period is [0,30), or 30 time units. We compute the left sub-tree's share of this local time partition's count as $\frac{(5-0)}{(30-0)} \cdot 15 = 2$, while the right sub-tree's share is $15 - 2 = 13$. In this case, the left sub-tree leaf node [0,5) now has a count of 2 (see figure 10(b)). We now pass 13 down the root right sub-tree, increasing its right leaf node count from [5,9)(50) to [5,9)(52) as its share of the newly added partition's count, 2, is added, by using the same proportion calculation method. At leaf node

*Table 2*. Leaf node values and resulting global partition in a tabular format once all 9 partitions from figure 9 are inserted.

| (a) Leaf nodes | | | (b) Resulting global partition ($p = 3$) | | |
|---|---|---|---|---|---|
| Count | Start | Stop | Count | Start | Stop |
| 17 | 0 | 5 | 95 | 0 | 28 |
| 64 | 5 | 9 | 95 | 29 | 866 |
| 3 | 9 | 10 | 95 | 866 | 1000 |
| 12 | 10 | 30 | | | |
| 44 | 30 | 350 | | | |
| 43 | 350 | 800 | | | |
| 21 | 800 | 1000 | | | |
| 40 | 1000 | 1500 | | | |
| 32 | 1500 | 5000 | | | |
| 9 | 5000 | 10000 | | | |

[9,800)(50), the inserted partition's count is now down to 11, since 2 was taken by node [5,9)(52).

Now, the second difference comes into play. Two new leaf nodes are created by splitting [9,800)(50). The new leaves are [9,30) and [30,800). Leaf [9,30) receives all the remaining inserted partition's count of 11. The count of 50 from [9,800)(50) is now divided up amongst the two new leaf nodes. The left leaf node receives $\frac{(30-9)}{(800-9)} \cdot 50 = 1$ of the 50, while the right leaf node receives 49. So the new left leaf node is now [9,30)(12), where 12 comes from $11 + 1$, and the new right leaf node shows as [30,800)(49). Again, see figure 10(b) for the result. Table 2(a) shows the leaf node values once all 9 local time partitions from figure 9 are inserted.

Now that the coordinator has the global span leaf counts and the optimal number of leaf nodes to be processed by each node, it can figure out the global partition set. For each node (except the last one), we continue adding the span leaf counts until it matches or surpasses the optimal number of leaf nodes. When the sum is more than the optimal number, we break up the leaf node that causes this sum to be greater than the optimal number, such that the leaf node count division is done in proportion to the period duration.

As an example, refer to Table 2(a). We know that the optimal number of periods per global partition is 95. We add the leaf node counts from the top until we reach node [10,30)(12). The sum at this point is 96, or 1 more than optimal. We break up [10,30)(12) into two leaf nodes such that the first leaf node period should contain a count of 11, and the newly created leaf node should contain only 1. Using the same idea of proportional count division, we can see that [10,28)(11) and [28,30)(1) are the two new leaf nodes. So the first global time partition has the period [0,28) and has a count of 95.

The computation for the second global time partition starts at [28,30)(1). Continuing on, the global time partitions for this example are shown in Table 2(b).

The reader should be aware that this global time partition resolution algorithm is not perfect. The actual number of local aggregation tree leaves assigned to each worker node

may not be identical. The reason is that the algorithm uses the local partition sets, which are just guides for the global partitioning (as they are equi-depth histograms). When a local partition has 50 leaf nodes in period [9,800), the global partition scheme assumes a uniform distribution within that partition, while the actual leaf nodes distribution may be heavily skewed.

We expect better scalability for TDM + C as compared to the SAT and SM algorithms because of the data redistribution and its load-balancing effect. However, there are two global synchronization steps, which may limit the performance obtained. First, all of the local partition sets must be completed before the global time set partitioning can begin. Second, all of the worker nodes must complete their merges and send their results to the coordinator before the client can receive the final result.

### *4.5. Time division merge (TDM)*

The fifth parallel algorithm, TDM, is identical to TDM + C, except that it has distributed result placement rather than centralized result placement. This algorithm simply eliminates the final coordinator results collection phase and completes with each worker node having a distinct piece of the final aggregation tree. A distributed result is useful when the temporal aggregate operation is a subquery in an enclosing distributed query (such as the correlated aggregate functions mentioned in Section 3). This allows further localized processing on the individual node's aggregation sub-result in a distributed and possibly more efficient manner.

## 5. Empirical evaluation

For the purposes of our evaluation, we chose the temporal aggregate operation `COUNT`, though the results should hold for all SQL aggregates. We performed a variety of performance evaluations on the seven parallel algorithms presented.

In all experiments, we measured wall clock time to finish. The aggregation trees for all experiments fit into main memory; no swapping of the aggregation tree to disk [9] was necessary.

### *5.1. Experimental environment*

The experiments were conducted on a 64-node shared-nothing cluster of 200 MHz Pentium machines, each with 128 MB of main memory and a 2 GB hard disk. Connecting the machines was a 100 Mbps switched Ethernet network, having a point-to-point bandwidth of 100 Mbps and an aggregate bandwidth of 2.4 Gbps in all-to-all communication.

Each machine was booted with version 2.4.2-2 of the Linux kernel. For message passing between the Pentium nodes, we used the LAM implementation of the MPI communication standard [2]. The version of LAM-MPI was 6.5.1.

### 5.2. Experimental parameters

We utilized synthetic data sets, for full control over various parameters, as well as a data set from a production application, specifically the personnel records system at the University of Arizona [7].

In the synthetic datasets, each tuple had three attributes, an SSN attribute (9 bytes) which was filled with random values, a StartDate attribute (16 bytes), and an StopDate attribute (16 bytes). The SSN attribute refers to an entry in a hypothetical employee relation. The StartDate and StopDate attributes were temporal instants which together denote a valid-time period. The data generation method varied from one experiment to another and is described later.

The *tuple size* was fixed at 41 bytes/tuple. The tuple size was intentionally kept small and unpadded so that the generated datasets could have more tuples before their size made them difficult to work with. Larger tuples would have increased the initial disk I/O but would not have affected subsequent processing, since only the timestamps of tuple are exchanged in all of these algorithms.

*NumProcessors* depends on the type of performance measurement. Scale-up and speed-up experiments used 2, 4, 8, 16, 32 and 60 processing nodes, while the variable reduction experiments used a fixed set of 32 nodes. Four of the 64 processors were experiencing hardware problems, and so were not used.

To see the effects of *data partitioning* on the performance of the temporal algorithms, the synthetic tables were partitioned horizontally either by SSN or by StartDate. The SSN and StartDate partitioning schemes attempted to model range partitioning [4] based on non-temporal and temporal attributes, respectively.

The total *database size* reflects the total number of tuples across all the nodes participating in a particular experiment run. For scale-up tests, the total database size increased with the number of processing nodes.

The amount of *data reduction* is 100 minus the ratio between the number of resulting leaves in the aggregation tree and the original number of tuples in the dataset,

$$Reduction(\%) = \begin{cases} 100 & \text{if } U = 2 \text{ and } A > 2 \\ 100(1 - U/A) & \text{otherwise,} \end{cases}$$

where $U \geq 2$ is the number of *unique* timestamps in the input dataset and $A \geq 2$ is the total number of timestamps in the input dataset. A reduction of 100 percent means that a 100-tuple dataset produces 1 leaf in the aggregation tree because all the tuples have identical StartDates and StopDates. The higher the reduction, the smaller the size of the aggregation tree, which means lower overhead in insertion.

This reduction can take place independently in each node (termed *local reduction*), or in the coordinating node (termed *global reduction*), or in both. The degree of local reduction will have a large impact on the performance of many of the algorithms, because it affects the amount of communication. We conjecture that the degree of global reduction will have a much smaller impact, as it won't affect the local processing nor communicating information either between processing nodes or to the central coordinator. For that reason, the global

reduction was fixed at 0% (that is, no reduction), with only the local reduction varied. A local reduction of 0% was achieved by ensuring that all the timestamps were unique.

The value of the *hole* parameter is the percentage of leaves that have a count of zero in an aggregation tree,

$$Hole(\%) = 100 \times E/L,$$

where $E$ is the number of leaves with the count of zero and $L$ is the total number of leaves in the aggregation tree. Hole is orthogonal to reduction; the value of hole could be nonzero even if there is no reduction. Unlike reduction, the hole parameter has no impact on the size of the aggregation tree. Holes in the local aggregation tree affect the volume of the data transfered on the network. Once the local aggregation tree is built, each worker only needs to keep the leaves of the aggregation tree. The worker discards the leaves with a count of zero because these leaves will not contribute to the final result. Removing holes has the advantage of reducing the number of leaves transfered on the network and thus the number of leaves to be merged later.

Similar to reduction, holes can occur independently in each node or in the coordinator. However, a global hole doesn't have any impact on data transfer. Therefore global holes are not considered in the rest of this paper. The local hole percentage in all the synthetic datasets was fixed at 50%.

### *5.3. Synthetic datasets*

We set up our first experiment to compare the scale-up properties of the proposed algorithms on a large dataset with no reduction. We used the measurements taken from this experiment as a baseline for later comparisons with subsequent experiments. Table 3 gives the parameters for this particular experiment.

***5.3.1. Baseline scale-up performance: No reduction/SSN partitioning.*** For this experiment, a synthetic dataset containing 3.8 M tuples was generated. Each tuple had a randomized SSN attribute and was associated with distinct periods of unit length (i.e., *StopDate* =

*Table 3*. Experimental parameters (baseline scale-up, no reduction, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 2, 4, 8, 16, 32, 60 |
| Tuple size in bytes | 41 |
| Tuples per processor | 65,536 |
| Total number of tuples | $p \times 65,536$ |
| Local reduction | 0 percent |
| Local hole | 50 percent |

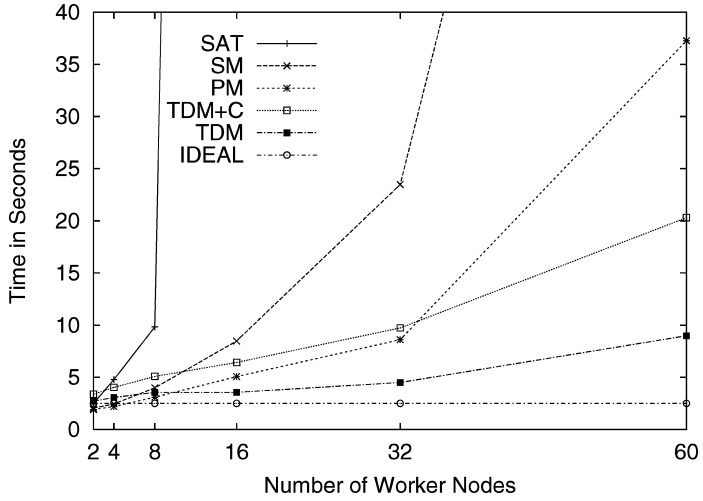



*Figure 11.* Experimental results (baseline scale-up, no reduction, SSN partitioning).

*StartDate* + 1). The dataset was sorted by SSN, then distributed to the processing nodes. Since the SSN fields were generated randomly, this had the effect of randomizing the tuples in terms of StartDate and StopDate fields.

To measure the scale-up performance of the proposed algorithms, a series of six runs having 2, 4, 8, 16, 32, and 60 nodes, respectively, was carried out. Note that since we fixed the size of the dataset on each node, increasing the number of processors meant increasing the total database size. The ideal, unrealizable algorithm would exhibit perfect scale-up: as the database size tracks the number of processors, the total time remains fixed. However, due to synchronization and communication costs, the algorithms do not exhibit ideal scale-up. Timing results from this experiment are plotted in figure 11 and lead us to the following conclusions.

*SM performs better than SAT.* Intuitively, since the dataset exhibits no reduction, both SM and SAT send *all* periods from the worker nodes to the coordinator. The reason behind SM's performance advantage comes from the computational parallelism provided by building local aggregation trees on each worker node. Aside from potentially reducing the number of leaves passed on to the coordinator, this process of building local trees sorts the periods in temporal order. The SM coordinator's use of a merge-sort variant in compiling the final result is more efficient than SAT's strategy of having to insert each valid-time period into the final aggregation tree.

*The performance difference between TDM and TDM + C increases with the number of nodes.* For this observation, it is important to remember that TDM + C is simply TDM plus an additional result-collection phase that sends all final leaves to the coordinator, one worker node at a time. The performance difference increases with the number of nodes because of the non-reducible nature of the dataset and the fact that scale-up experiments work with more data as the number of nodes increase.

*PM outperforms TDM + C up to 32 nodes.* We attributed this to the multiple merge levels needed by PM. A PM computation needs at least lg $p$ merge levels. On the other hand, the TDM + C algorithm only merges local trees once but has three synchronization steps, as described in Section 4.4. With this analysis in mind, we expected PM to perform better or as well as TDM + C for 2, 4, and 8 nodes, which have 1, 2, and 3 merge levels, respectively. We then expected TDM + C to outperform PM as more nodes are added, but we were surprised to realize that PM was still performing better than TDM + C up to 32 nodes.

To find out what was going on behind the scenes, we used the LAM XMPI package [2] to visually track the progression of messages within the various TDM + C and PM runs. This led us to the reason why TDM + C performed worse than PM for 2 to 32 nodes: TDM + C was slowed more by increased waiting time due to load-imbalance (computation skew) as compared to PM.

*SAT exhibits the worst scale-up performance.* This result is not surprising, since the only advantage SAT has over the original sequential algorithm comes from parallelized I/O. This single advantage does not make up for the additional communication overhead and the coordinator bottleneck, as all the periods are sent to the coordinator which builds a single, but large, aggregation tree. Since SAT is not competitive, we will not consider it in the following experiments.

TDM exhibits the best scale-up performance. The difference between the performance of TDM and the ideal line is stable before 32 nodes, and it gets larger after 32 nodes. This is caused partially by the limitation of our network environment. We will discuss this in detail in Section 9.2. This network limitation impacts the performance when the algorithm runs on 32 and 60 processors in all test cases. We will see this impact in almost all the figures in the rest of this paper. Even under this situation TDM shows a good scale-up performance. For 60 nodes, that is, a data set 60 times larger than the sequential data set, the total time was only 5 times that of the much smaller sequential data set.

***5.3.2. Speed-up performance: No reduction/SSN partitioning.*** Unlike the scale-up experiment, the data size was fixed in this experiment at $60 \times 65536$ tuples $= 3932160$ tuples. This data set is partitioned by SSN and distributed evenly to a varying number of nodes participating in the experiment. As in the last experiment, the local reduction here is also 0%. The experimental parameters for this are given in Table 4 (we don't repeat the tuple size, as it is constant over all experiments, unless stated otherwise). We hoped that the overall time would drop by half when going from 8 to 16 processors, then by half again to

*Table 4.* Experimental parameters (speed-up, no reduction, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 8, 16, 32, 60 |
| Tuples per processor | 491,520/245,760/122,880/65,536 |
| Total number of tuples | $60 \times 65,536$ |
| Local reduction | 0 percent |

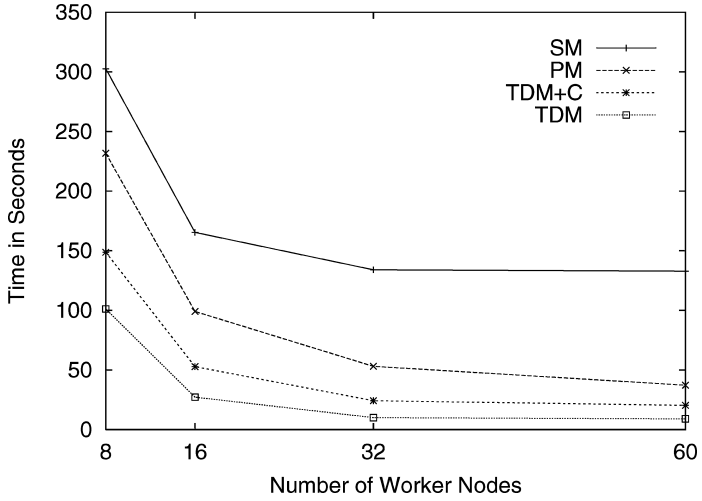



*Figure 12*. Experimental results (speed-up, no reduction, SSN partitioning).

32 processor and again by almost half in going to 60 processors. Timing results from this experiment are given in figure 12.

*All but SM have good speedup property.* As the number of processing nodes increased, the performance of all the algorithms was improved. We observed the performance was improved significantly when the number of nodes changed from 8 to 16 and from 16 to 32. However, when the number of nodes increased to 60, the performance didn't improve significantly. One reason is there are more workers to be synchronized. Another reason is the network limitation we have mentioned in the last section. Although, the performance improvement is still observed, it is not as much as that in the previous cases.

***5.3.3. Scale-up performance: 100% reduction/SSN partitioning.*** This experiment was designed to measure the effect of a significant amount of reduction (100% in this case) on the scale-up properties of the proposed algorithms. Table 5 gives the parameters for this experiment. This experiment was modeled after the first one but with a synthetic dataset having 100% (local) reduction. This dataset was generated by associating all tuples on

*Table 5*. Experimental parameters (scale-up, 100% reduction, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 2, 4, 8, 16, 32, 60 |
| Tuples per processor | 327,680 |
| Total number of tuples | $p \times 327{,}680$ |
| Local reduction | 100 percent |

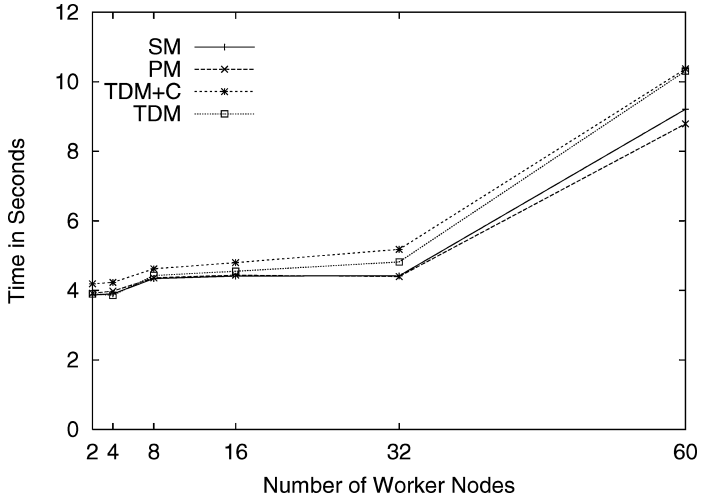



*Figure 13.* Experimental results (scale-up, 100% reduction, SSN partitioning).

each node with the same period (for complete local reduction); the SSN attribute values were random. The high reduction gives us the opportunity to use a larger data set and still guarantee that the aggregation trees fit in the main memory. Here we use 327,680 tuples per node. The results are shown in figure 13.

*All algorithms benefit from the 100% data reduction.* Comparing results from the baseline experiment with results from the current experiment lead us to this observation. Because of the high degree of data reduction, the aggregation trees do not grow as large as in the first experiment. With smaller trees, insertions of new periods take less time because there are fewer branches to traverse before reaching the insertion points. Because all of the presented algorithms use aggregation trees, they all experience increased performance.

*With 100% reduction, PM and TDM + C catch up to TDM.* Aside from constructing smaller aggregation trees, a high degree of data reduction decreases the number of aggregation tree leaves exchanged between nodes. TDM does not send its leaves to a central node for result collection, so it does not transfer as many leaves as its peers. Because of this, TDM is not improved by the amount of data reduction as much as either PM or TDM + C which end up performing as well as TDM.

Note that even in the worst case, 4 M tuples on 60 processors using TDM-C takes only 2.5 times longer than 650 K tuples on two processors.

***5.3.4. Speed-up performance: 100% reduction/SSN partitioning.*** This experiment was designed to measure the effect of high data reduction on the speed-up properties of the proposed algorithms. The data size was fixed in this experiment at 60 $\times$ 327,680 tuples = 19,660,800 tuples. The local reduction is 100%. This data set is distributed evenly to a varying number of nodes participating in the experiment. The experimental parameters for this are shown in Table 6. Timing results from this experiment are plotted in figure 14.

*Table 6*. Experimental parameters (speed-up, 100% reduction, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 8, 16, 32, 60 |
| Tuples per processor | 2,457,600/1,228,800/614,400/327,680 |
| Total number of tuples | $60 \times 327{,}680$ |
| Local reduction | 100 percent |

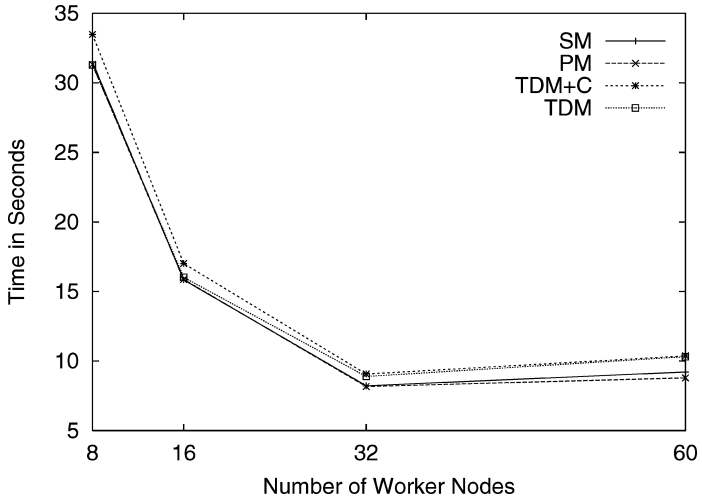



*Figure 14*. Experimental results (speed-up, 100% reduction, SSN partitioning).

*With 100% reduction, all the algorithms show good speedup property up to 32 nodes.* The performance of all the algorithms is slightly worse at 60 nodes than that at 32 nodes. The results of the last section illustrate the cost of synchronization is relatively high when the data reduction is high. Again, the degradation of the performance at 60 nodes is caused partially by the network limitation.

***5.3.5. Performance with variable reduction/SSN partitioning.*** This experiment was designed to measure the effect of a varying amount of data reduction on the scale-up properties of the proposed algorithms. Six datasets with different reduction were generated. The experiment setting is provided in Table 7 and timing results are plotted on figure 15. Note that the values plotted for a reduction of 0% correspond to those plotted in figure 11 for 32 nodes.

*As the reduction increases, the performance improves.* Since the reduction implies the amount of tuples with same timestamp in datasets, as the reduction increases, so does the number of identical tuples. Hence, the aggregation tree does not grow as much and performance improves.

*Table 7.* Experimental parameters (variable reduction, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 32 |
| Tuples per processor | 65,536 |
| Total number of tuples | $32 \times 65{,}536 = 2{,}097{,}152$ |
| Local reduction | 0/20/40/60/80/100 percent |

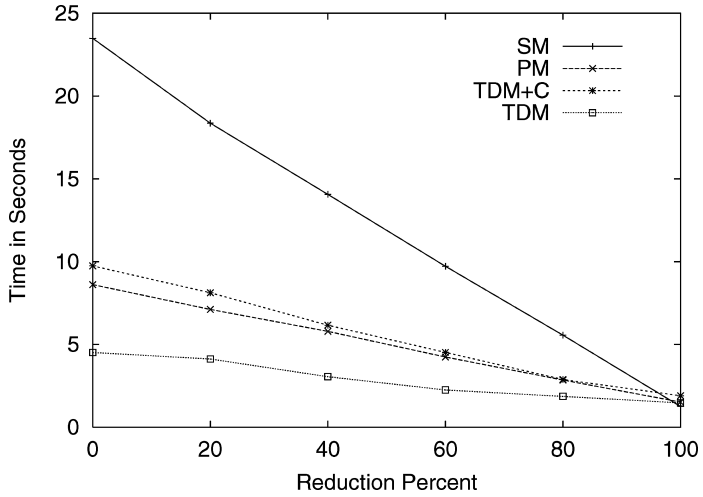



*Figure 15.* Experimental results (variable reduction, SSN partitioning).

***5.3.6. Scale-up performance: No reduction/time partitioning.*** The previous experiments all assumed SSN partitioning (note however that the algorithms don't know of this partitioning). We now turn our attention to timestamp partitioning.

This first experiment was designed to measure the effect of time partitioning on the scale-up properties of the proposed algorithms. The dataset for this experiment was generated in a manner similar to the baseline experiment, but with StartDate rather than SSN partitioning. This was done by sorting the dataset by the StartDate attribute and then distributing it to the processing nodes. The experimental settings are summarized in Table 8 and the timing results are provided in figure 16.

*Time Partitioning did not significantly help any of the algorithms.* We originally expected TDM and TDM + C to benefit from the time partitioning but we also realized that for this to happen, the partitioning must closely match the way the global time divisions are calculated. Because we randomly assigned partitions to the nodes, TDM did not benefit from the time partitioning. Sometimes it performed a little bit better while in other cases it performed a little bit poorer (compare with figure 11). We attribute the small performance gaps to differences in how the partitioning strategies interacting with the number of nodes made

*Table 8.* Experimental parameters (scale-up, no reduction, time partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | Time |
| Number of processors ($p$) | 2, 4, 8, 16, 32, 60 |
| Tuples per processor | 65,536 |
| Total number of tuples | $p \cdot 65{,}536$ |
| Local reduction | 0 percent |

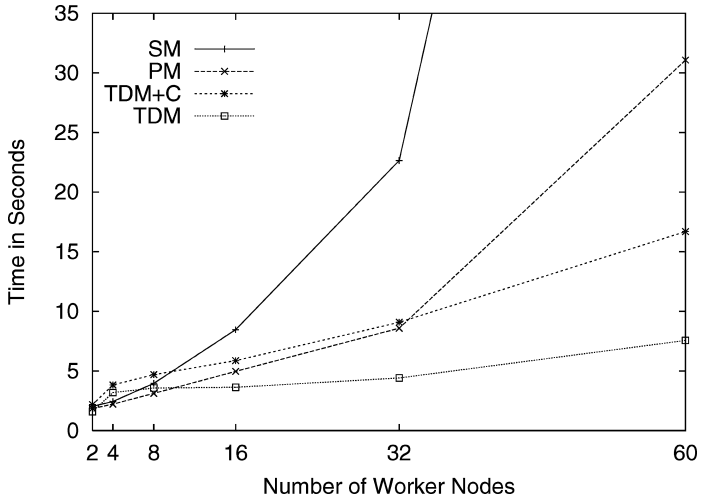



*Figure 16.* Experimental results (scale-up, no reduction, time partitioning).

TDM redistribute mildly varying numbers of leaves across the runs. As for SM and PM, they exhibited no conclusive improvement because they were simple enough to work without considering how tuples were distributed across the various partitions.

***5.3.7. Speed-up performance: No reduction/time partitioning.*** The experimental parameters for this are shown in Table 9 and results from this experiment are plotted in figure 17. Like scale-up performance, the speed-up performance in this section is very similar to that in Section 5.3.2. Time partitioning simply doesn't make much difference.

***5.3.8. Performance with variable reduction/time partition.*** For this experiment, six sets of partitions were generated. Each set had 32 partitions, one for each of the 32 processing nodes participating in the six runs. The partitions were generated having 0, 20, 40, 60, 80 and 100 percent reduction. The settings for this experiment, provided in Table 10, summarizes the parameters for this experiment. Timing results for this experiment are plotted on figure 18.

*Increasing the amount of data reduction improved the performance of the proposed algorithms.* Like in Section 5.3.5, increasing the amount of reduction improved the performance

*Table 9.* Experimental parameters (speed-up, no reduction, time partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | Time |
| Number of processors ($p$) | 8, 16, 32, 60 |
| Tuples per processor | 491,520/245,760/122,880/65,536 |
| Total number of tuples | $60 \times 65{,}536$ |
| Local reduction | 0 percent |

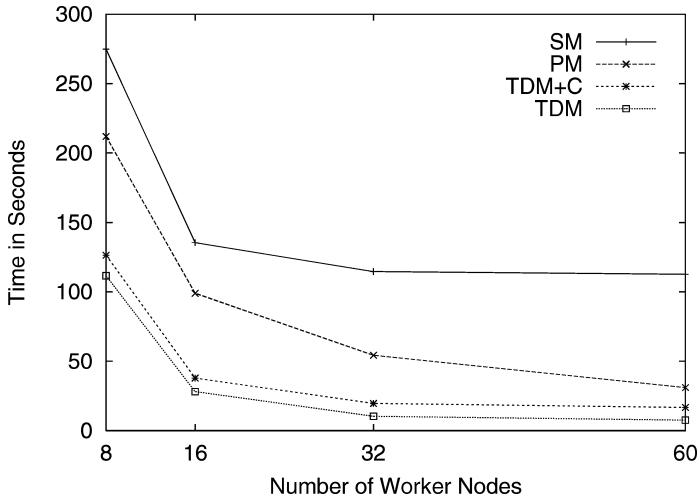



*Figure 17.* Experimental results (speed-up, no reduction, time partitioning).

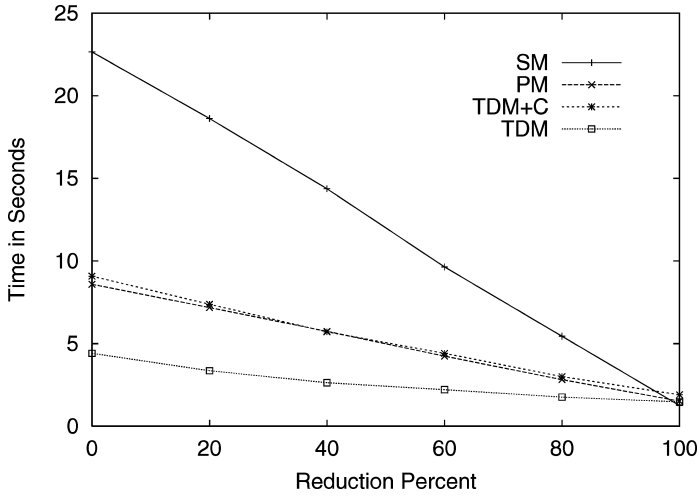



*Figure 18.* Experimental results (variable reduction, time partitioning).

*Table 10.* Experimental parameters (variable reduction, time partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | Time |
| Number of processors | 32 |
| Tuples per processor | 65,536 |
| Total number of tuples | $32 \times 65{,}536 = 2{,}097{,}152$ |
| Local reduction | 0/20/40/60/80/100 percent |

of the parallel algorithms. With higher degrees of data reduction, aggregation trees became increasingly smaller with fewer leaves to exchange between nodes.

*Varying data reduction doesn't significantly affect TDM.* The flat slope of TDM's performance curve in figure 18 as compared with figure 15 shows us that this is the algorithm that is least affected by variations in local reduction. The reason for this is that, among the presented algorithms, TDM exchanges the least number of leaves as discussed when we observed that the performance for TDM + C and PM caught up with TDM for 100% local reduction.

## 6. A real-world dataset

Tuples in the synthetic datasets used in the experiments to this point have timelines of unit length, which is not realistic. For this next set of experiments, we applied the count aggregate to a salary table drawn from the University of Arizona's personnel system, termed the *UIS dataset* [7]. For this dataset, the reduction, tuple size, and database size was necessarily fixed, at 83,857 tuples and 7.8 Mbytes. For this reason, we used a maximum of 32 processors. Also, the UIS dataset used in this experiment has tuples with timelines of variable length. Note that this data set exhibited quite high reduction, at 80.76%, indicating that many timestamps are duplicates. For example, an employee changing jobs would have a starting time for the new position identical to the ending time of the old position.

### *6.1. Scale-up performance: SSN partitioning*

This experiment was designed to measure the scale-up properties of the proposed algorithms on the UIS dataset partitioned by SSN. The dataset was sorted by SSN and distributed to the processing nodes. A varying number of nodes was used, thus applying the aggregate over a varying percentage of the database size (for example, for two nodes, only one-sixteenth of the data was used). The experimental parameters for this are shown in Table 11, and the results are plotted in figure 19.

Since the data set is quite small, the absolute difference between the performance of different algorithms is also small. However, we can see SM and PM behaves better than TDM and TDM + C at 32 nodes. With high degree of data reduction and small data set, the algorithms that need less synchronization cost perform better.

*Table 11.* Experimental parameters (scale-up, SSN partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | SSN |
| Number of processors ($p$) | 2, 4, 8, 16, 32 |
| Tuple size in bytes | 93 |
| Tuples per processors | 2,620 |
| Total number of tuples | $p \cdot 2{,}620$ |
| Local reduction | 80.76 percent |

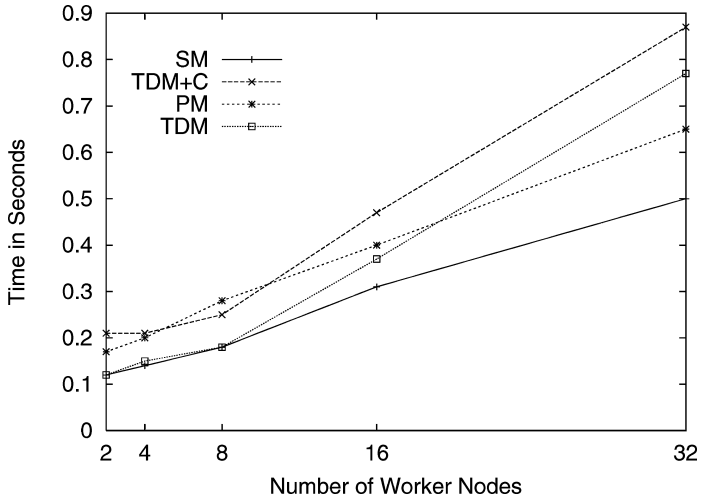



*Figure 19.* Experimental results (scale-up, SSN partitioning).

### *6.2. Scale-up performance: Time partitioning*

The experimental parameters for this are shown in Table 12, with the results given in figure 20. We can observe analogous results to the same experiment on dataset partitioned by SSN (cf. figure 19). Just like the results of several experiments on synthetic datasets partitioned by time, these experiments with realistic data continue to show that time partitioning doesn't greatly affect the scale-up of the proposed algorithms.

## 7. Better node assignment

We initially expected that the performance of TDM would improve markedly when the data was partitioned by time, but that performance advantage did not materialize. The reason is that TDM arbitrarily assigns partitions to processors and so doesn't exploit any clustering of the data by time.

*Table 12.* Experimental parameters (scale-up, time partitioning).

| Parameters | Actual values |
|---|---|
| Partitioning | Time |
| Number of processors ($p$) | 2, 4, 8, 16, 32 |
| Tuples per processor | 2,620 |
| Total number of tuples | $p \cdot 2{,}620$ |
| Local reduction | 80.76 percent |

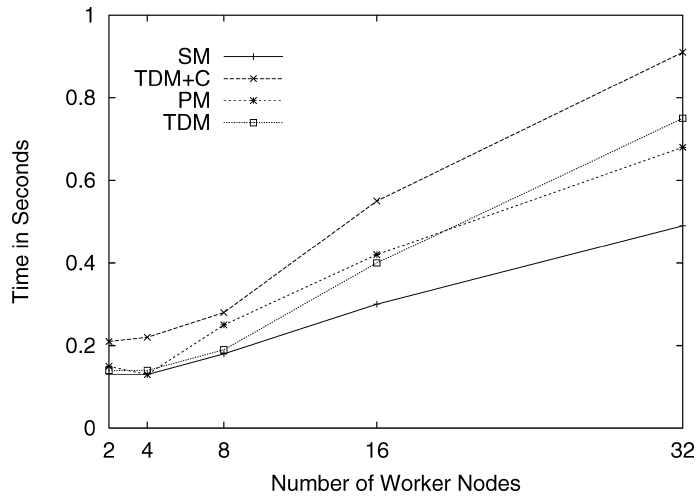



*Figure 20.* Experimental results (scale-up, time partitioning).

Specifically, in TDM and TDM + C we assign global partitions to worker nodes in a naive manner. This assignment policy may cause large data movements among workers especially when the partitioning doesn't match the way the global time divisions are calculated. For illustration, consider a dataset partitioned into $p$ workers in two different ways. In the first partitioning, each time division $i$ matches exactly the timeline of the dataset of worker $i$. In the second partitioning, time division $i$ doesn't match the timeline of the dataset of worker $i$. If we run TDM on these two cases, the performance will be different because of the assignment policy of TDM. There will be no data movement in the first case, and large data movement in the second.

We now turn to two new algorithms, Greedy Time Division Merge (GTDM) and Greedy Time Division Merge with Centralization (GTDM + C), which are improved versions of TDM and TDM + C, respectively.

### *7.1. Greedy time division merge with centralization (GTDM + C)*

This algorithm is another variant of TDM + C, improving the performance by utilizing a better global partitioning assignment policy that attempts to minimize the number of

leaves redistributed. In Greedy Time Division Merge the coordinator receives the local division from each worker and calculates the global division. For each global division, the coordinator computes the number of overlaped tuples in each worker. The worker node that has the maximal number of overlaped tuples is assigned to this division, and is not considered for the remaining global divisions. This greedy allocation is repeated until every global division is assigned.

The performance gap between TDM and GTDM on the two kinds of partitioning is shown in figure 21. Timeline match means the assignment of TDM and the assignment of GTDM are the same. Everything will be same except that GTDM has an extra overhead on the greedy assignment. However, this overhead is less than 3% of the total time as shown in Section 9. Timeline mismatch indicates TDM has to exchange all the data on the network

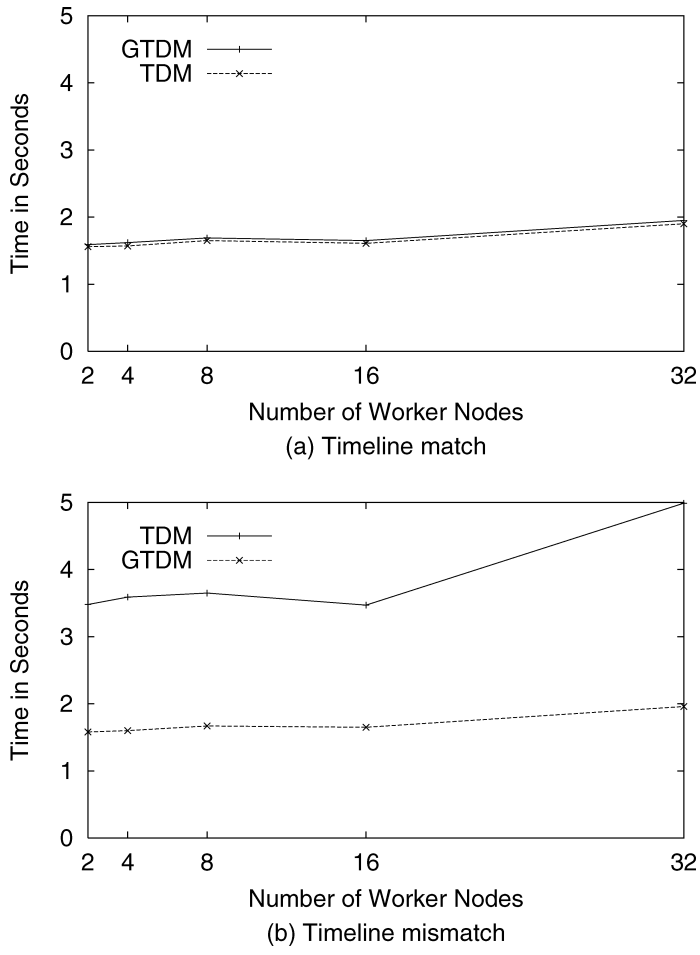


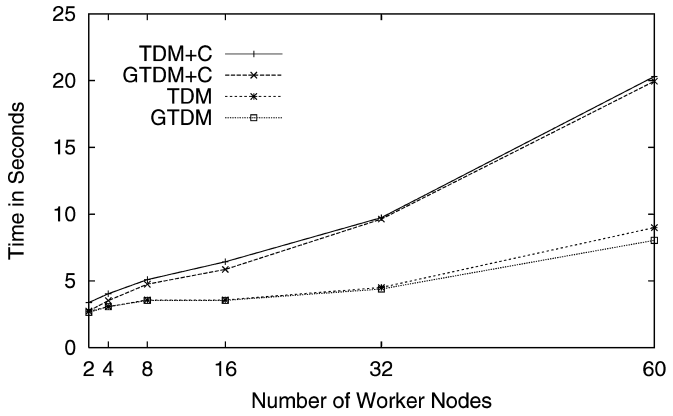

*Figure 21.* Performance gap between GTDM and TDM.

and merge the data later, while GTDM doesn't need to do any of them. Since exchanging data and merging this data take the majority of the total time as shown in Section 9, GTDM pays much less than it gains in the overall performance.

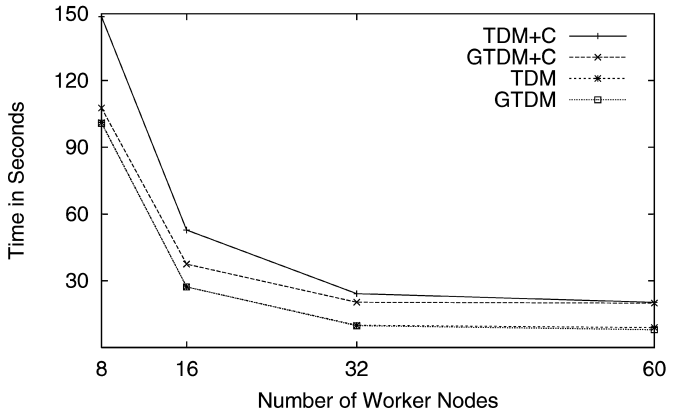
GTDM is identical to GTDM + C except that it doesn't collect final results; in this way, it is analogous to TDM.

We now compare GTDM and TDM, and their centralized counterparts, by performing the same experiments as those done previously. We expected little improvment on SSN partitioned data, but increased performance on time-partitioned data.

*GTDM + C performs slightly better than TDM + C when running on SSN partitioned data with no local reduction.* When data is partitioned by randomly generated SSN, the timeline covered by each worker may overlap in many places. So, even if we apply the greedy assignment policy, we can't avoid data movements. However, as shown in figure 22(a),




(a) Scale-up, No Reduction, SSN Partitioning, Synthetic dataset




(b) Speed-up, No Reduction, SSN Partitioning, Synthetic dataset

*Figure 22.* Experimental results (synthetic dataset with SSN partitioning, no reduction).

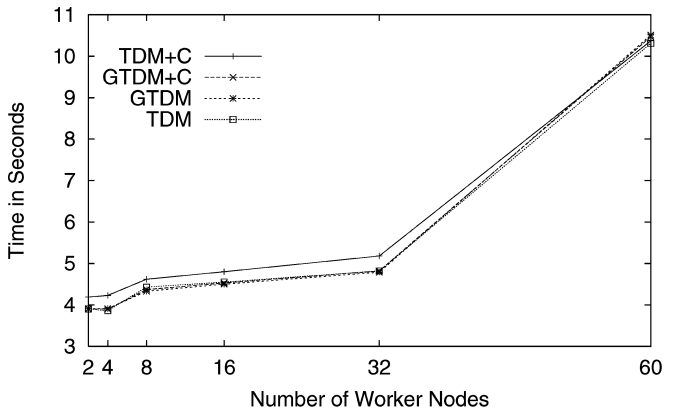
we can observe slight performance improvement even in this case because of minimized network traffic caused by the greedy assignment policy.

*GTDM and TDM show similar speedup property for SSN partitioning with no reduction.* The results are show in figure 22(b). As the number of nodes decreases, the difference between the performance of GTDM + C and that of TDM + C becomes bigger.

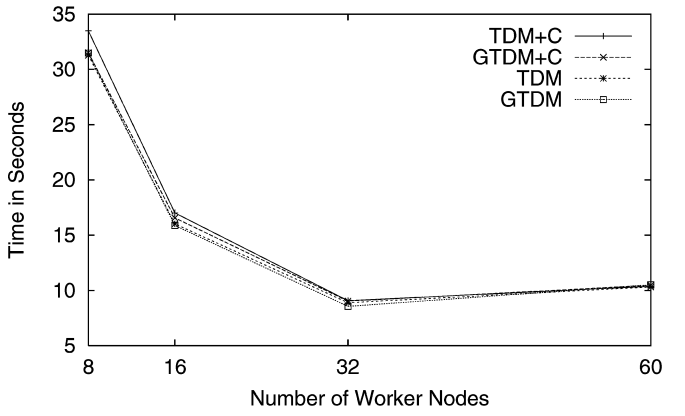
*GTDM also benefits from 100% data reduction for SSN partitioning.* Since number of leaves in the aggregation tree in each worker is small because of 100% reduction, GTDM also takes advantage of this (figure 23(a)).

*With 100% reduction, results are similar.* See figure 23(b).

*As the reduction increases, the performance improves.* Since the reduction implies the amount of tuples with same timestamp in datasets, as the reduction increases, so does the number of identical tuples (figure 24).




(a) Scale-up, 100% Reduction, SSN Partitioning, Synthetic dataset




(b) Speedup, 100% Reduction, SSN Partitioning, Synthetic dataset

*Figure 23.* Experimental results (synthetic dataset with SSN partitioning, 100% reduction).

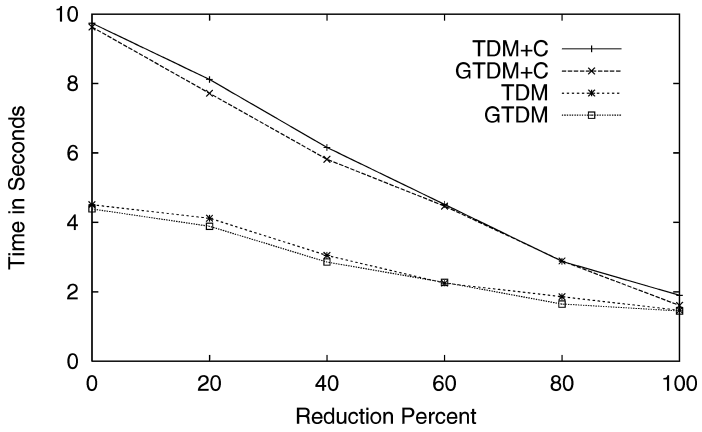



*Figure 24.* Experimental results (synthetic dataset with variable reduction, SSN partitioning).

We now turn to time partitioning.

*GTDM can outperform TDM when the dataset is time partitioned.* The dataset used in this experiment is partitioned by time and there is no reduction in it. So, there is no overlap among timelines covered by each worker. Since we assigned each partition of the dataset to each worker in a random manner, the timeline of worker $i$ is not equal to the time division $i$ of global partition. Consequently, data movement among workers happened in TDM, but in GTDM such data movement didn't occur because of the balanced assignment policy (figure 25(a)). In the case of 60 processors, the overall time for GTDM was only 2.9 times optimal.

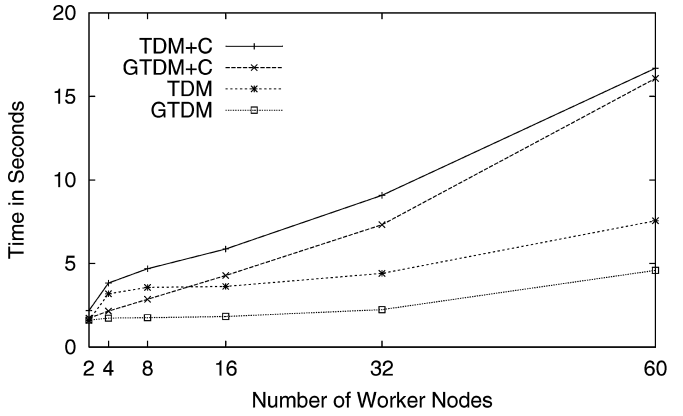
*As the number of nodes decreases, the performance of the greedy algorithms improves significantly with time partitioning.* When the number of nodes decreases, the number of tuples in each node increases. The greedy assignment policy benefit the most from time partitioning, while TDM probably need to exchange all the tuples on one node with all on another. At 8 and 16 nodes, the centralized GTDM + C even outperforms TDM (figure 25(b)).

*Varying data reduction doesn't significantly affect GTDM under time partitioning.* The gentle slope of GTDM's performance curve in figure 26 shows us that it is the algorithm least affected by variations in local reduction.

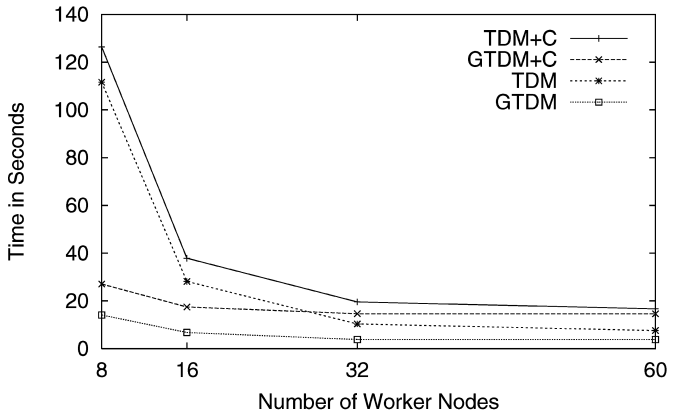
*Increasing the amount of data reduction improves the performance of GTDM.* As we've already observed in the previous experiment, increasing the amount of reduction improved the performance of the parallel algorithms. With higher degrees of data reduction, aggregation trees became increasingly smaller with fewer leaves to exchange between nodes.

*GTDM outperforms TDM slightly with the real dataset and SSN partitioning.* Since the real dataset is too small, the greedy assignment doesn't show an obvious improvement on the performance (figure 27(a)).

*GTDM outperforms TDM with the real dataset with time partitioning.* The results shown in figure 27(b) are similar to the results with synthetic data.




(a) Scale-up, No Reduction, Time Partitioning, Synthetic dataset




(b) Speed-up No Reduction, Time Partitioning, Synthetic dataset

*Figure 25.* Experimental results (synthetic dataset with time partitioning, no reduction).

## 8. Summary of experiments

The empirical observations confirm that dataset partitioning, result placement, data reduction effected by the aggregation, and the number of processing nodes all affect the performance of the proposed algorithms, in different ways. SAT and SM, as seen in figures 11 and 16, were affected most by the number of processing nodes. Figure 18 shows that SM, SAT, PM and TDM + C were significantly slowed by low data reduction while TDM was the least affected. Also, figures 11, 13, and 16 show that TDM has the best performance under all situations, but only if distributed result placement is desired. On the other hand, PM has centralized result placement but is superior to TDM + C only in two small areas of the parameter space: high reduction and large configurations (figure 13) and low reduction and small configurations (figures 11 and 16). Dataset partitioning only affected the TDM

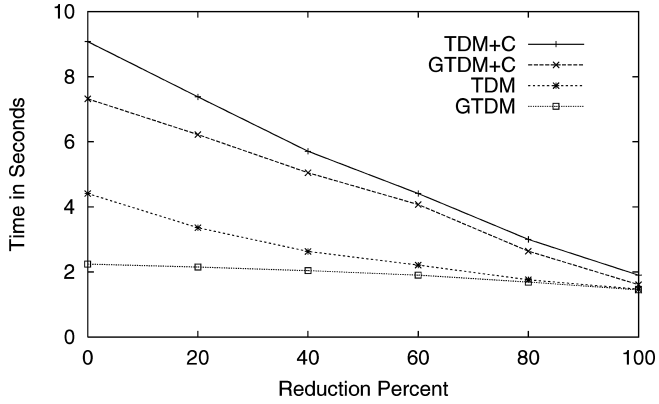



*Figure 26*. Experimental results (synthetic dataset with variable reduction, time partitioning).

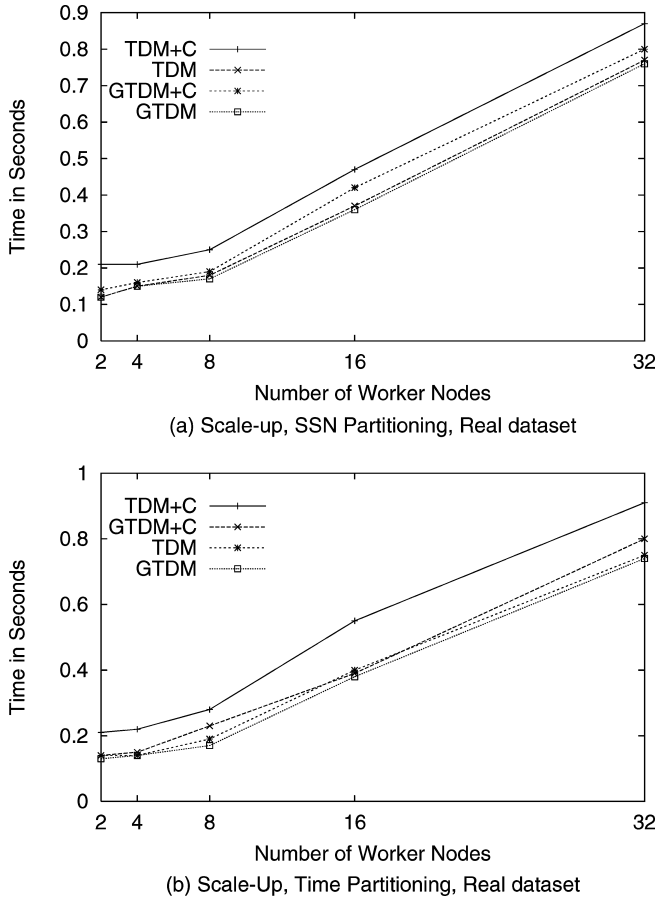



*Figure 27*. Experimental results (scale-up with real dataset).

*Table 13.* Time of major steps of GTDM.

| Steps | Time (secs) | Time/total time (%) |
|---|---|---|
| Build local aggreation tree $t_L$ | 1.6 | 22 |
| Calculate local partition set | 0.017 | 0.23 |
| Calculate global partition set | 0.20 | 2.8 |
| Assign global partition set | 0.030 | 0.41 |
| Exchange data $t_E$ | 3.1 | 43 |
| Merge data locally $t_M$ | 2.3 | 32 |

variants, and even then, not substantially (compare figure 11 with figure 8). These results parallel those for the real-world dataset (figures 19 and 20).

GTDM (and GTDM + C) has virtually the same performance as TDM (and TDM + C) for SSN partitioning (figure 22). For time partitioning, GTDM differs from TDM with low reduction. GTDM + C differs from TDM + C with low reduction *and* small to medium configurations. GTDM (and GTDM + C) performs much better than TDM (and TDM + C) when the data set size on each node is large. In all cases, the greedy variant was superior.

## 9. Cost model

We now introduce a cost model to predict the performance of GTDM. The major components of the response time in GTDM are listed in Table 13. The data set used in Table 13 is the same synthetic data set used in Table 3 with the number of processors set at 60.

The times to calculate the local partition set, to calculate the global partition set, and to assign the global partition set are all less than 3% of the total time, and are thus negligable. In Section 9.1, we will discuss the cost of the other three components, which comprise the bulk of the total response time. This is shown in the following formula,

$$t = t_L + t_E + t_M,$$

where $t$ is the total response time while $t_L$, $t_E$, and $t_M$ are the time of building local aggregation tree, exchanging data, and merging data, respectively.

### *9.1. The cost of each step*

The cost of building the local aggregation tree $t_L$ includes three components: reading tuples from the disk, insert tuples in the aggregation tree, and propagating the aggregation values to the leaves. The time to read tuples from the disk is $t_r \cdot s \cdot n/B$, where $t_r$ is the time per sequential read, $s$ is the tuple size, $n$ is the number of tuples in each worker, and $B$ is the block size in bytes. The time to insert tuples in the aggregation tree depends on the data set.

The number of leaves in the aggregation tree is

$$l = \begin{cases} 2n(1-R) & \text{if } R < 1 \\ 3 & \text{if } R = 1, \end{cases}$$

where $R$ is the local reduction of the data set. In the worst case, where the timestamps are sorted, the aggregation tree is a linear list. The time complexity of building the aggregation tree is $O(n \cdot l)$. However, when the time stamps are in random order, the tree is balanced and the time complexity of building aggregation tree is $O(n \lg l)$. To propagate the aggregation values to the leaf nodes, the whole tree is traversed once. The time complexity is $O(n(1-R))$. Hence, the cost formula of building local aggregation tree is as follows.

$$t_L = \begin{cases} \dfrac{t_r s n}{B} + c_1 n \lg l + c_2 n(1-R) & \text{if time is in random order} \\ \dfrac{t_r s n}{B} + c_1' n l + c_2 n(1-R) & \text{otherwise} \end{cases}$$

$c_1$, $c_1'$ and $c_2$ are constants in units of seconds. We will show how these constants are calculated in the next section.

The cost of exchanging data $t_E$ depends on two parameters: the amount of data transfered and the data transfer speed. Since GTDM uses a greedy strategy to assign partitions to nodes, in each worker at most $(p-1)/p$ of the total leaves of the local aggregation tree need to be sent to other workers. Each worker receives the same amount of data it sends out. The cost of exchanging data is $2s_l l(1-H)(p-1)/(p \cdot T)$, where $s_l$ is the size of a leaf of the aggregation tree, $H$ is the fraction of local holes (the leaves representing the holes are thrown away before exchanging data), and $T$ is the throughput per worker. In the algorithm, each worker sends a fixed size of data ($s_p$ leaves as one packet) to another worker at one time. So the cost of exchanging data is

$$t_E = \frac{2 s_l s_p}{T} \left\lceil \frac{l(1-H)}{p \cdot s_p} \right\rceil (p-1).$$

The last component $t_M$ is the cost of merging the leaves. The leaves sent from one worker are sorted by time. The most efficient method is multiway merge, which has the time complexity of $O(\sum_{i=1}^{m} n_i \lg m)$, where $n_i$ is the number of leaves in each list and $m$ is the number of lists. However, our program merges the lists one by one, for simplicity, which has the cost of

$$t_M = c_3 \left( n_1(m-1) + \sum_{i=2}^{m} n_i(m+1-i) \right),$$

where $c_3$ is a constant in units of second. Each worker sends $s_p$ leaves to another worker at one time. These $s_p$ leaves make up one list in the destination worker. The first list in each

worker comes from itself with the length of $l(1-H)/p$. So, we have

$$n_i = \begin{cases} \dfrac{l(1-H)}{p} & \text{if } i = 1 \\ s_p & \text{otherwise} \end{cases}$$

$$m = (p-1)\left\lceil \frac{l(1-H)}{p \cdot s_p} \right\rceil + 1.$$

When the data set is time-partitioned, the assignment of GTDM guarantees there is no data transferred. Therefore, the cost of merging data is zero.

### *9.2. The parameters and constants*

The values of some of the parameters appearing in the cost model depend on the data set used in different experiments. These parameters have been described in Section 5. Other parameters depend only on the experimental environment and the implementation. The values of these parameters in our experiments and implementation are shown in Table 14.

Why do we have a function over the number of processors, $T(p)$, for the network throughput? The throughput is determined by the network equipment connecting the processors. In Section 5.1, we mentioned that the network connecting the processors has a point-to-point bandwidth of 100 Mbps and an aggregate bandwidth of 2.4 Gbps in all-to-all communication. Theoretically, when there are less than 24 processors sending data to the network, the throughput per processor is stable. The bandwidth available to each processor goes down along with the increasing of the number of processors when more than 24 processors send data to the network at the same time. Since our programs are running on MPI, we expect that the actual throughput of transfering useful data in our program can't reach the ideal maximum bandwidth.

To find out the value of $T$ for our configuration (a Cisco ethernet switch and the MPI runtime library), we ran two programs that transfer data in different styles. In the first program, the $i$th processor sends 600 big packets to the $(i+1)$ processor, and then receives the same amount of data from the $(i-1)$ processor. The $p$th processor sends to the first processor and then receives the same amount of data from the $p-1$ processor. The size of each packet is 12.8 KB. All processors send the data at the same time. We ran this program on different number of processors. The results are shown by the pt2pt line in figure 28. As we expected, the throughput per processor is stable before the network is saturated. After

*Table 14.* Some of the parameters in the cost formula.

| Description | Parameter | Value |
|---|---|---|
| Leaf size | $s_l$ | 16 bytes |
| Packet size | $s_p$ | 800 values |
| Block size | $B$ | 1 Kbyte |
| Sequential block read | $t_r$ | 0.148 msecs |
| Throughput per processor | $T$ | $T(p)$ |

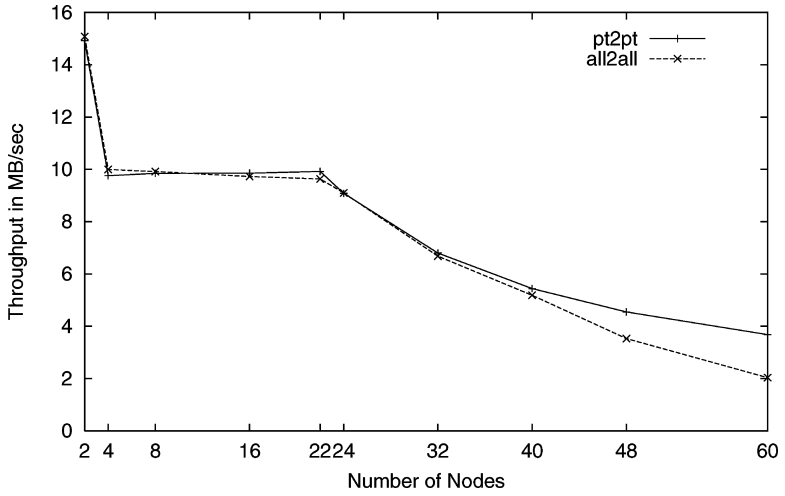



*Figure 28.* Throughput per processor.

the number of processors is greater than 22, the thoughput per processor goes down by the factor of $1/p$. The elbow is 22 instead of 24 because there are some overhead added by the software and network protocol between MPI program and the low level network.

However, when the data transfer style is changed, the network throughput changes, sometimes significantly. In the second program, each processor sends $600/(p-1)$ packets to every other processor and receives the same amount of data from every other processor. For example, the $i$th processor sends $600/(p-1)$ packets to the $(i+1)$ processor and receives data from the $(i-1)$ processor. The $i$th processor next sends $600/(p-1)$ packets to the $(i+2)$ processor and receives data from the $(i-2)$ processor, and so on. The results are shown by the all2all line in figure 28. The throughput for the second program tracks that of the point-to-point program up through 32 processors. After this point, the throughput drops faster than in the first case. This indicates there are some other resource limitations in the network that caused the network congestion. One fact is the first half of our cluster is configured to be able to communicate in full duplex, and the second half are configured to be able to communicate only in half duplex (due to limitations with our switch). When the number of processors passed 32, more and more half duplexed processors participated in the communication. In the point-to-point transfer style, since each processor only talks to its neighbor, a half-duplexed processor only impacts the throughput of two communications. While in the all-to-all transfer style, since each processor talks to every other processors, a half-duplexed processor will impact the throughput of all the communications. Another possible limitation is the size of the buffer inside the switch. When the buffer is full, congestion occurs. In all-to-all communication, when the number of processors increases, the possibility of congestion increases.

The GTDM algorithm uses the all-to-all communication style, so we employ that measured throughput in our cost model, as $T(p)$.

This fall-off in network throughput as the number of processors increases can explain why the performance of our algorithms was not as effective when the number of processors

*Table 15.* Constants in the cost formula.

| Constants | Values (*usecs*) | $R^2$ |
|---|---|---|
| $c_1$ | 0.896 | 0.9995 |
| $c_2$ | 2.30 | 0.9995 |
| $c_3$ | 0.525 | 0.9751 |

increased from 32 to 64. This is the reality of the network environment the algorithms ran on. The behavior of the network may not be the same if the execution environment is different. For example, a switched Ethernet network with the nonblocking architecture won't have the limitation of the aggregated bandwidth. The throughput between any two processors doesn't change along with the changes of the number of processors. This would cause our algorithms to perform better when the number of processors is higher.

The remaining constants in the cost model are determined by studying the architecture of the processors. We compute these constants by doing linear regressions of the time and the parameters. For example, $c_1$ appears as an coefficient in the cost of inserting tuples to the local aggregation tree. We ran a number of tests each of which has different value of $n$. The times of inserting tuples to the local aggregation tree were collected. The coefficient $c_1$ was computed by doing linear regression of the time and the corresponding $n \lg l$. Other constants are computed in the same way. Table 15 shows the value of the constants and the corresponding coefficient of determination $R^2$. A covariance very close to one indicates that tree insertion is indeed $n \lg n$. We didn't compute $c_1'$ since the time stamps are in random order in the synthetic data set in our experiments.

### *9.3. Estimating the performance*

All the parameters and constants in our environment are now available. We can use the cost formula to estimate the performance of GTDM. The estimated results for scale-up experiments on SSN partitioned data set are shown in figure 29.

The estimated time matches the actual time of GTDM well when the number of processors is less than 32. The difference between the estimated time and the actual time at 60 processors is 2.09 seconds. To find out the reason, we ran some tests to calculate the actual throughput of our program. The results show the actual throughput of our program is lower than the all-to-all throughput shown in figure 28, but the behavior of the throughput in the two situations are similar. The reason is when we test network throughput, the program only involves communication, while the program of GTDM has to prepare the data before sending them out and allocate memory for the newly arrived data after each receiving. Therefore, the actual throughput of our program is lower than the all-to-all throughput. The gap between the estimated time and the real time is mainly due to this.

The estimated results are similar to the actual results in all the other cases. We show some of the results in figures. Figure 30 shows the results for speed-up experiments on SSN partitioned data set. Figure 31 shows the results for variable reduction experiments on SSN partitioned data set.

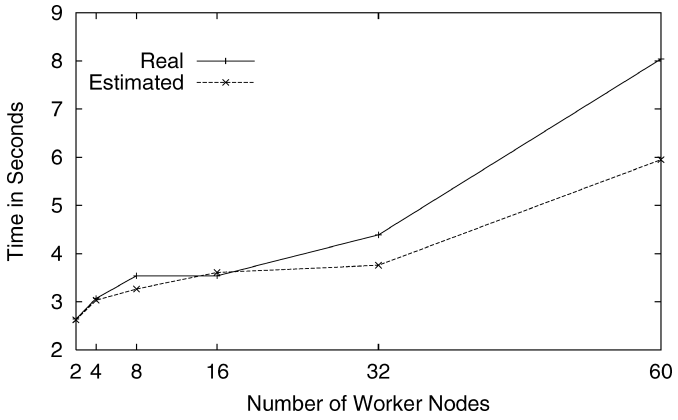



*Figure 29.* Scale-up, no reduction, SSN partitioning, synthetic dataset.

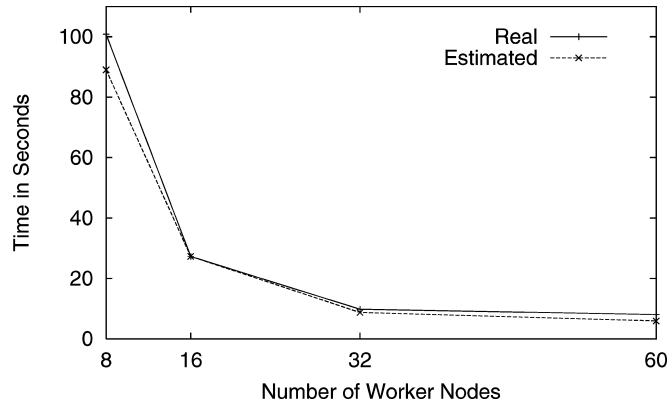



*Figure 30.* Speed-up, no reduction, SSN partitioning, synthetic dataset.

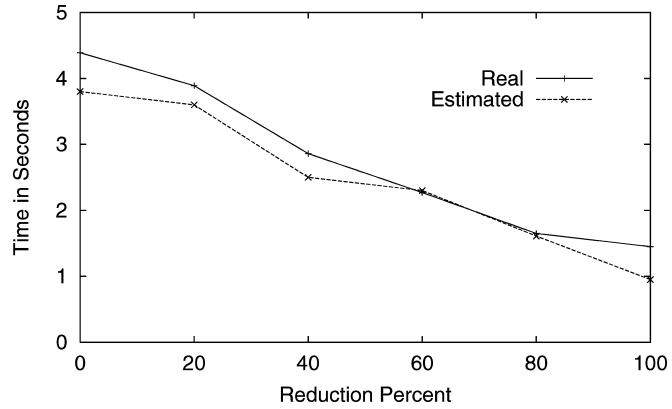



*Figure 31.* Variable reduction, SSN partitioning, synthetic dataset.

*Table 16*. Matrix of recommendations.

| Data reduction | Node count | Distributed result | Centralized result |
|---|---|---|---|
| HI | Small | GTDM | GTDM + C |
| | Large | GTDM | PM |
| LOW | Small | GTDM | PM (SSN), GTDM + C (Time) |
| | Large | GTDM | GTDM + C |

## 10. Conclusions

Temporal aggregate computations are important operations in a temporal database system. Traditionally, this has been an expensive operation in sequential database systems, which don't, as yet, use the aggregation tree. Therefore, the question arises as to whether parallelism is a cost-effective approach for improving the efficiency of temporal aggregate computations.

The main contribution of this paper is a collection of novel algorithms that parallelize the computation of temporal aggregates. We ran these algorithms through a series of experiments to observe how different properties affected their performance. Finally, we developed and validated a cost model for GTDM that closely predicts the running time of this algorithm.

From these observations, we provide the following conclusions which should help in the design of a parallel database system's query optimizer that selects the right temporal algorithm for a particular situation. Our recommendations are summarized in the matrix in Table 16.

1. Use GTDM whenever distributed result placement suffices, regardless of any other parameter. (This only applies when the manner in which the result is distributed is appropriate. Otherwise, it is probably best to go with a centralized result, which can then be redistributed as desired.) As discussed in Section 3, distributed result placement is useful for distributed sub-queries which are parts of larger distributed queries. Also, distributed result placement suffices when the aggregation results are not required for the *entire* time line (e.g., finding the (time-varying) salaries of all employees for the last year).
2. For centralized result placement, use PM only when there is a high degree of data reduction and large configuration or when there is a relatively low data reduction, a small configuration and the data is time partitioned.
3. Otherwise, for centralized result placement, use GTDM + C.

If one were to implement only one algorithm, our recommendation would be to choose GTDM, with an optional collection step.

Our experimental observations lead us to the following issues for future research. In a temporal aggregate query with tuple placement and/or selection skew, some worker nodes will complete their local aggregation tree faster than other nodes. We expect PM to outperform TDM + C in queries with heavy tuple placement skew and/or selection skew [19]. However, the specific impact of skew should be investigated.

Uneven computing time on the processing nodes as caused by dataset characteristics and system load make nodes unnecessarily wait idly for more loaded nodes. Strategies such as the opportunistic merging in PM for balancing the loads among the nodes would help reduce idle-waiting and improve the performance of the other algorithms. Several of the algorithms used equi-depth histograms to attempt to estimate the workload at each processing node; perhaps this estimate can be improved. In addition, we assume a homogenous architecture. Load balancing is more challenging on a heterogeneous parallel machine containing processors of different capability and performance.

Our proposed algorithms rely solely on main memory for storing runtime information, which include merged lists, aggregation trees and, message queues. A disk-paging strategy that is aware of how the parallel algorithms work [9] would allow the algorithms to handle larger dataset sizes.

We have studied the effects of different parameters on the proposed algorithms. Other factors such as long-lived tuples and data distribution may affect the performance of the algorithms.

Finally, we have focused here on scalar aggregates, which return a single result at each point in time. It would be interesting to extend these approaches to accommodate grouping, such as "the (time-varying) maximum salary *per department*."

## Acknowledgments

This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037), NSF Grants IIS-0100436, CDA-9500991, EAE-0080123, IRI-9632569, IIS-9817798, and NSF Research Infrastructure Programs EIA-0080123 and EIA-9500991, and by grants from Amazon.com and the Boeing Corporation. We would like to thank the reviewers for their suggestions, which improved this paper. We also thank Greg Andrews and Tom Lowry for their helpful insights on the impact of the network architecture on our performance measures.

## References

1. D. Bitton, H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Parallel algorithms for the execution of relational database operations," ACM Transactions on Database Systems, vol. 8, no. 3, pp. 324–353, 1983.
2. Ohio Supercomputer Center, LAM/MPI Parallel Computing. `http://www.osc.edu/lam.html`, 1998.
3. D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The gamma database machine project," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, pp. 44–62, 1990.
4. D.J. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," Communications of the ACM, vol. 35, no. 6, pp. 85–98, 1992.
5. R. Epstein, "Techniques for processing of aggregates in relational database systems," Technical Report UCB/ERL M7918, University of California, Berkeley, CA, Feb. 1979.
6. J.C. Freytag and N. Goodman, "Translating aggregate queries into iterative programs," in Proceedings of the 12th VLDB Conference, Kyoto, Japan, 1986, pp. 138–146.
7. J.A.G. Gendrano, R. Shah, R.T. Snodgrass, and J. Yang, "University information system (UIS) dataset," Technical Report TIMECENTER CD-1, Department of Computer Science, University of Arizona, Sept. 1998.

8. C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes, and S. Jajodia (Eds.), "A glossary of temporal database concepts," ACM SIGMOD Record, vol. 23, no. 1, pp. 52–64, 1994.
9. N. Kline, "Aggregation in temporal databases," PhD dissertation, Computer Science Department, University of Arizona, May 1999.
10. N. Kline and R.T. Snodgrass, "Computing temporal aggregates," in Proceedings of the IEEE International Conference on Data Engineering, Taipei, Taiwan, March 1995, pp. 222–231.
11. B. Moon, I.F. Vega López, and V. Immanuel, "Scalable algorithms for large temporal aggregation," in Proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, 2000, pp. 145–154.
12. M. Muralikrishna and D.J. DeWitt, "Equi-depth histograms for estimating selectivity factors for multi-dimensional queries," in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1988, pp. 28–36.
13. R.T. Snodgrass and I. Ahn, "Temporal databases," IEEE Computer, vol. 19, no. 9, pp. 35–42, 1986.
14. R.T. Snodgrass, S. Gomez, and E. McKenzie, "Aggregates in the temporal query language TQuel," IEEE Transactions on Data Engineering, vol. 5, pp. 826–842, 1993.
15. M. Stonebraker, "The case for shared nothing," A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering, vol. 9, no. 1, pp. 4–9, 1986.
16. Transaction Processing Performance Council (TPC), TPC Benchmark D (Decision Support), Standard Specification, Revision 1.3.1, Aug. 1998.
17. P.A. Tuma, Implementing Historical Aggregates in TempIS, 1992. Master's Thesis.
18. C. Turbyfill, C. Orji, and D. Bitton, "AS$^3$AP: An ANSI SQL standard scaleable and portable benchmark for relational database systems," in J. Gray (Ed.), The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann Publishers, 1991, chapter 4, pp. 167–207.
19. C.B. Walton, A.G. Dale, and R.M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in Proceedings of the VLDB Conference, Barcelona, Spain, Sept. 1991, pp. 537–548.
20. J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," in Proceedings of the IEEE International Conference on Data Engineering, Heidelberg, Germany, 2001, pp. 51–60.
21. J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," Very Large Databases Journal, vol. 12, no. 3, pp. 262–283, 2003.
22. X. Ye and J.A. Keane, "Processing temporal aggregates in parallel," in IEEE International Conference on Systems, Man, and Cybernetics, Orlando, FL, Oct. 1997, pp. 1373–1378.
23. D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger, "Efficient computation of temporal aggregates with range predicates," in Proceedings of the ACM Principles of Database Systems, Santa Barbara, CA, May, 2001, pp. 118–126.