Parallel Aggregation for Temporal Databases

Bongki Moon Jose Alvin G. Gendrano Minseok Park
Richard T. Snodgrass Bruce C. Huang Jim M. Rodrigue

TR-42

A TimeCenter Technical Report

Title	**Parallel Aggregation for Temporal Databases**
Author(s)	Bongki Moon Jose Alvin G. Gendrano Minseok Park Richard T. Snodgrass Bruce C. Huang Jim M. Rodrigue
Publication History	September 1999. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Renato Busatto, Curtis E. Dyreson, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Bongki Moon, Sudha Ram

Individual participants
Anindya Datta, Georgia Institute of Technology, USA
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, State University of Campinas and EMBRAPA, Brazil
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium
Carlo Zaniolo, University of California at Los Angeles, USA
Nick Kline, Microsoft, USA

For additional information, see The TIMECENTER Homepage:
URL: `<http://www.cs.auc.dk/TimeCenter>`

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

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 shared-nothing architectures; these algorithms are based on the sequential Aggregation Tree algorithm. 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.

1 Introduction

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 [15]; all but one of the 17 TPC-D benchmark queries involve aggregates [13]. Hence, efficient execution of aggregate functions is an important goal.

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.

Name	Salary	Begin	End
Richard	40K	18	∞
Karen	45K	8	20
Nathan	35K	7	12
Nathan	37K	18	21

(a) Data Tuples

Count	Begin	End
1	7	8
2	8	12
1	12	18
3	18	20
2	20	21
1	21	∞

(b) Result of Count

Max	Begin	End
35K	7	8
45K	8	20
40K	20	∞

(c) Result of Max Salary

Table 1: Sample Database and Sample Temporal Aggregations

In this paper, we present several new parallel algorithms for the computation of temporal aggregates on shared-nothing architectures [12]. Specifically, we start with the (sequential) Aggregation Tree algorithm [9] 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.

This paper is organized as follows. Section 2 gives a review of related work and presents the sequential algorithm on which we base our parallel algorithms. Our proposed algorithms on computing parallel temporal aggregates are then described in Section 3. Section 4 presents empirical results obtained from the experiments performed on a shared-nothing Pentium cluster. Finally, Section 5 concludes the paper and summarizes future work.

2 Background and 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 extended Epstein's algorithms to handle temporal aggregates [11]; these were further extended by Tuma [14] and by Kline [8, 9]. The resulting algorithms were quite effective in a uniprocessor environment. However, because they are inherently sequential, they all suffer from poor scale-up performance, which identifies the need to develop parallel algorithms for computing temporal aggregates.

Early research on developing parallel algorithms focused on the framework of general-purpose multiprocessor 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. More recently, parallel algorithms for handling temporal aggregates were presented [17], but for a shared-memory architecture.

The parallel temporal aggregation algorithms proposed in this paper are based on the (sequential) Aggregation Tree algorithm (SEQ) designed by Kline [9]. The 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 end time, and a count. When an aggregation tree is initialized, it begins with a single node containing $< 0, \infty, 0 >$ (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. The start time value, 18, of the first entry to be inserted splits the initial tree, resulting in the updated aggregation tree shown in Figure 1. Because the original node and the new node share the same end date of ∞, a count of 1 is assigned to the new leaf node $< 18, \infty, 1 >$. The aggregation tree after inserting the rest of the records in Table 1(a) is shown at the bottom of Figure 1.

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 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)).

Though SEQ correctly computes temporal aggregates, it is still a sequential algorithm, bounded by the resources of a single processor machine. This makes a parallel method for computing temporal aggregates desirable.

3 Parallel Processing of Temporal Aggregates

In this section, we propose seven 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

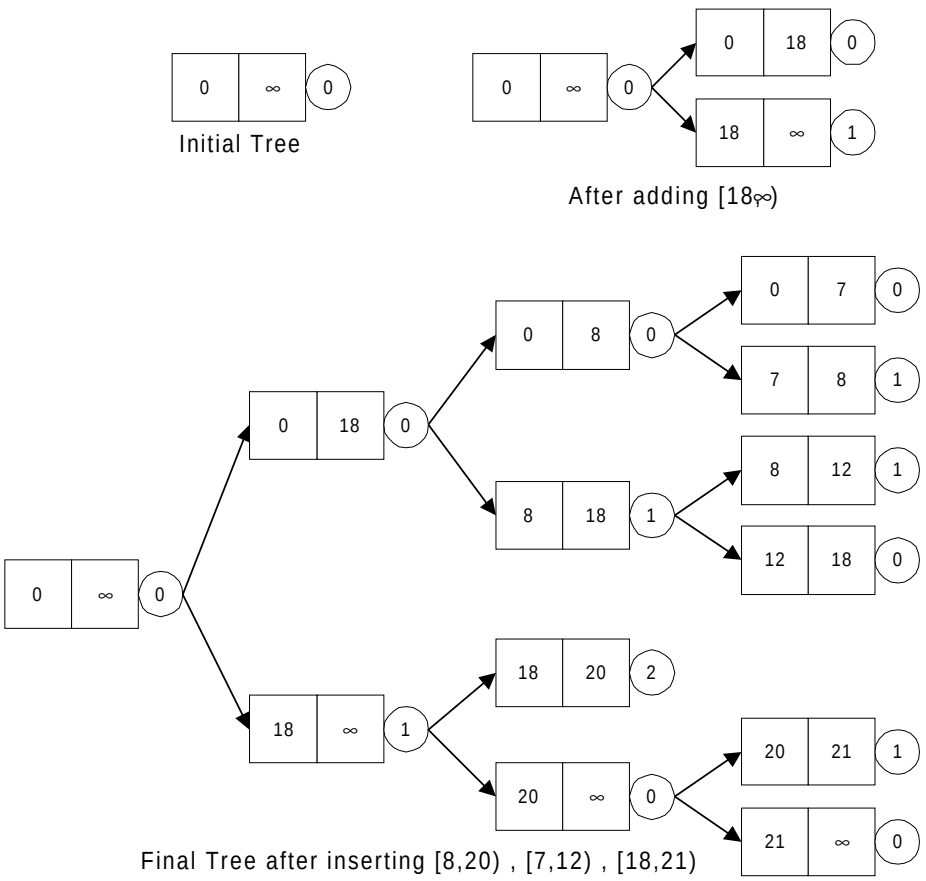


Figure 1: Example Run of the Sequential(SEQ) Aggregation Tree Algorithm, for Count

processors, as differentiated from the centralized results produced by the other algorithms. Finally, we present Greedy Time Division Merge (GTDM) and Greedy Time Division Merge with Centralization (GTDM+C), which are improved versions of TDM and TDM+C, respectively, for minimized communication overhead.

3.1 Single Aggregation Tree (SAT)

The first algorithm, SAT, extends the Aggregation Tree algorithm by parallelizing disk I/O. 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. While SAT can exploit disk I/O parallelism, the entire task of aggregate computation is accomplished solely by the coordinator sequentially.

3.2 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

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

Figure 2: Major Steps for the Pairwise Merge Algorithm

any parent nodes are discarded. 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 (no aggregation tree is constructed at the coordinator). The use of this efficient merging algorithm is possible since 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. However, the task of merging leaves is still carried out by the coordinator sequentially, which will be a limiting factor for the scalability of the SM algorithm.

3.3 Pairwise Merge (PM)

The third parallel algorithm, Pairwise Merge (see Figure 2), attempts to alleviate the sequential bottleneck of the SM algorithm by parallelizing the task of merging leaves. Worker nodes are paired to merge their leaves in each of $\log_2 p$ local synchronization steps, instead of merging all the leaves by the coordinator node. Which two worker nodes are paired in each local synchronization step is determined dynamically by the query coordinator.

Each 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. 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 3 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 near linear in the number of leaf nodes to be merged.

3.4 Time Division Merge with Centralization (TDM+C)

Like PM, the fourth parallel algorithm, TDM+C, also extends the aggregation tree method to exploit both computational and I/O parallelism (see Figure 4). Moreover, TDM+C attempts to achieve even higher degree of parallelism by enabling all the worker nodes to participate in the task of merging leaves in parallel. The main steps for this algorithm are outlined in Figure 5.

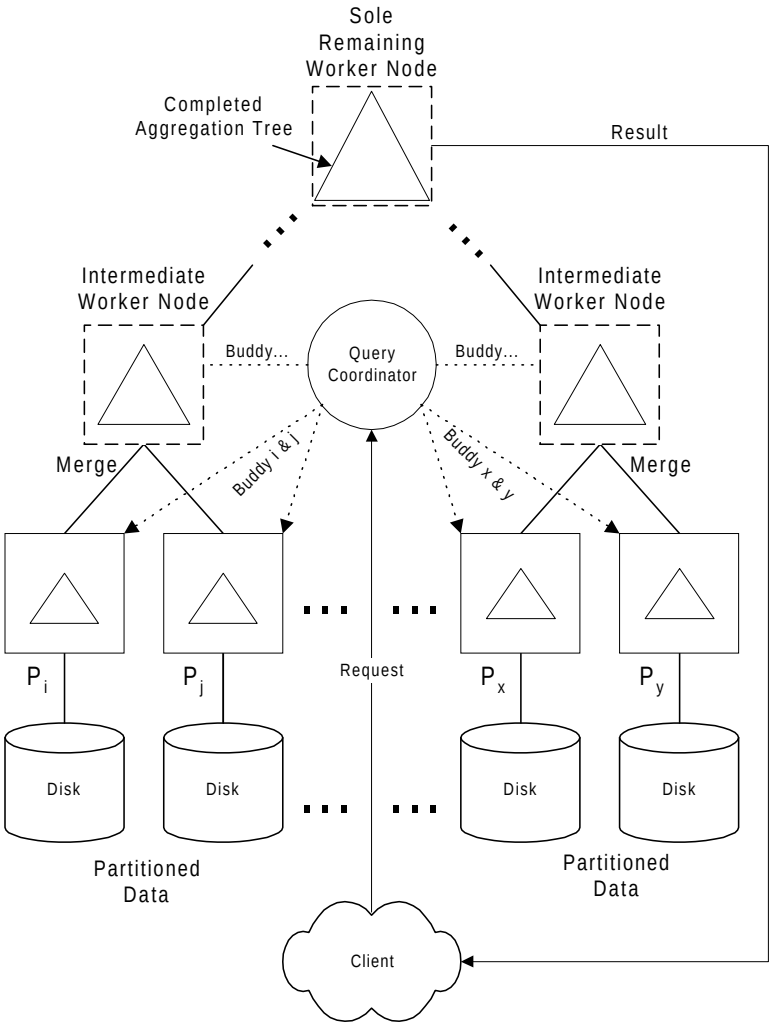


Figure 3: Pairwise Merge (PM) Algorithm

3.4.1 Global Timeline Division

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 timestamp and maximum (latest) end timestamp values to ascertain the overall timeline of the query. The timeline then will be divided into p partitions in a way that the task of merging leaves is evenly distributed across p worker nodes.

The task of dividing timeline largely relies on the idea of equi-depth histogramming [10], and the task is performed in two steps. First, each worker node divides its local timeline into p partitions such that each partition can match almost the same number of local leaves. Of course, for the reason, the partitions can have different lengths of durations. This timeline division is essentially an equi-depth histogram for the set of local leaves. Second, the local timeline partitions from each worker node are sent to the coordinator, which then computes a global timeline division from a total of p^2 local partitions (how this is done is discussed in detail in the next section). The resulting timeline division is a global p-interval equi-depth histogram across the entire data set.

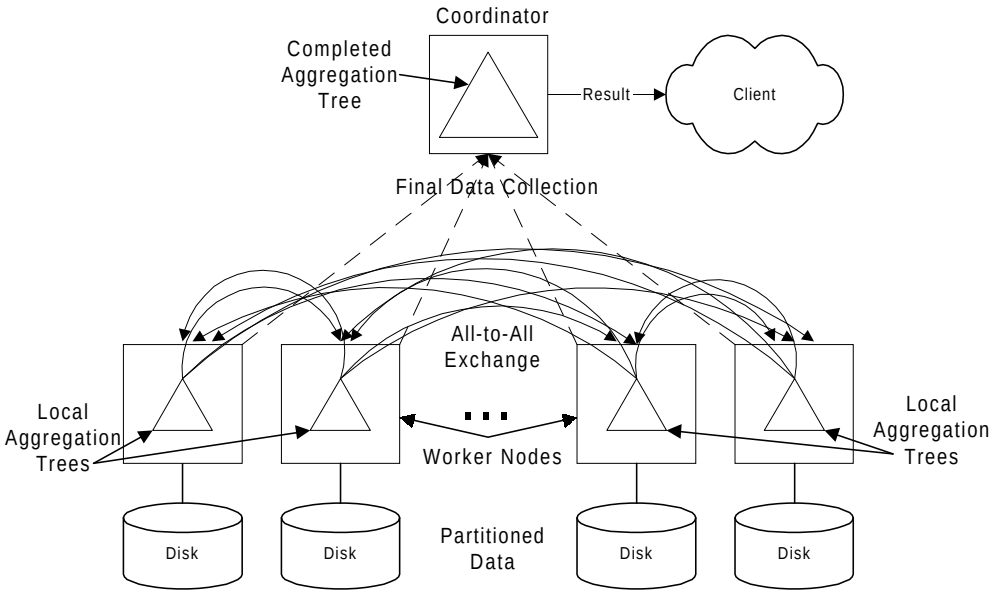


Figure 4: Time Division Merge with Centralizing Step (TDM+C) Algorithm

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 results to client

Figure 5: Major Steps for the TDM+C Algorithm

After computing the global timeline division, the coordinator broadcasts the global timeline division to all the worker nodes. Then, the i^{th} worker node will take up the i^{th} partition for the rest of parallel execution of aggregation. Each worker node uses this information of global timeline division 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 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.

3.4.2 Calculating the Global Partition Set

We examine in more detail the computation of the global partition set by the coordinator. 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.

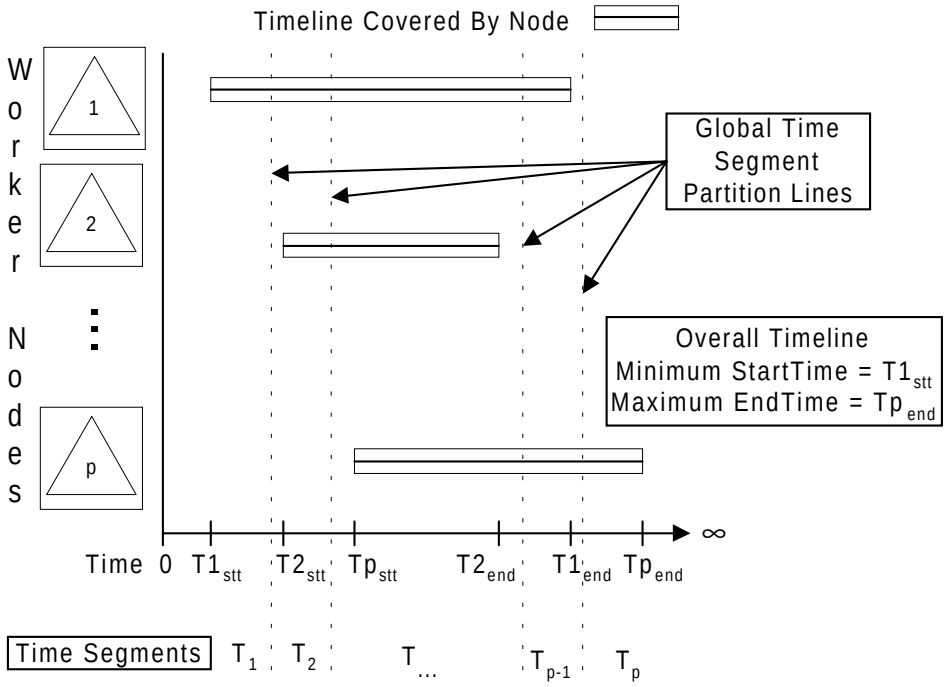


Figure 6: Timeline divided into p partitions, forming a global partition set

As an example, Figure 7 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 nodes is $50 \times 3 + 15 \times 3 + 30 \times 3 = 285$. The best plan is having $\frac{285}{3} = 95$ leaf nodes to be processed by each node. Figure 6 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 sets as periods, inserting them into the modified aggregation tree. From Figure 7, the first period to be inserted is [5,9)(50), the fourth is [0,30)(15), and 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.

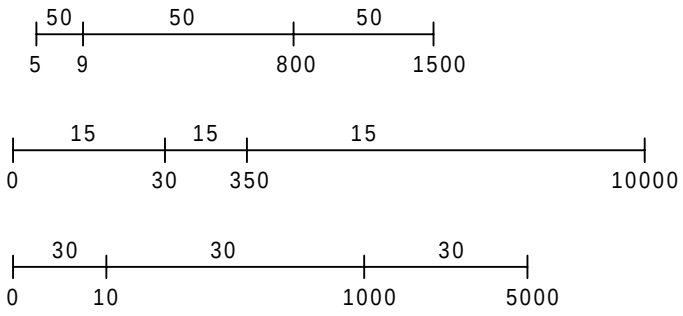
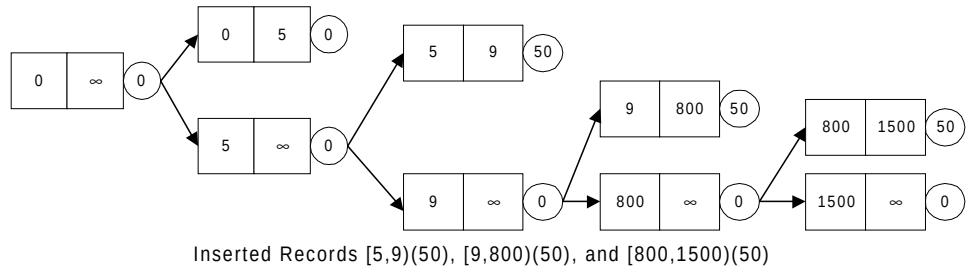
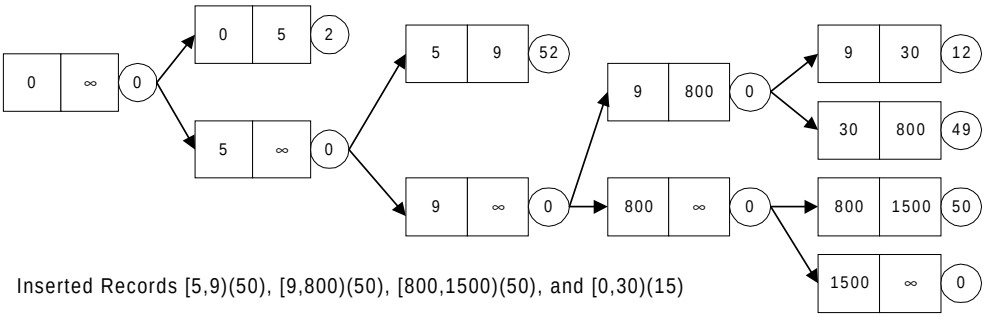


Figure 7: Local Partition Sets from Three Worker Nodes



(a) First 3 Local Partitions



(b) After partition 4 is added

Figure 8: Intermediate Aggregation Tree

There are three main differences between our Modified Aggregation Tree algorithm used in this portion of TDM+C and the original Aggregation Tree [9] used in step 2 of Figure 5. 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, a parent node must have a count value of 0. When a leaf node is split and becomes a parent node, its count is split proportionally between the two new leaf nodes based on the durations of their respective time periods. This new parent count 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 8a. 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 the 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 $\frac{(5-0)}{(30-0)} \times 15 = 2$, 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 8b). 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.

Count	Begin	End
17	0	5
64	5	9
3	9	10
12	10	30
44	30	350
43	350	800
21	800	1000
40	1000	1500
32	1500	5000
9	5000	10000

(a) Leaf nodes

Count	Begin	End
95	0	28
95	29	866
95	866	1000

(b) Resulting global partition ($p = 3$)

Table 2: Leaf node values and resulting global partition in a tabular format once all 9 partitions from Figure 7 are inserted

At leaf node [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 divvied up amongst the two new leaf nodes. The left leaf node receives $\frac{(30-9)}{(800-9)} \times 50 = 1$ 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 8b for the result. Table 2 shows the leaf node values once all 9 local time partitions from Figure 7 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 2a. 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 2b.

It should be noted that this global timeline division algorithm may not be able to achieve perfect load balance at all times. The reason is that the algorithm relies on approximate information of local load distributions summed up in 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 with the SAT, SM and PM algorithms, provided that an input database is evenly distributed across the worker nodes. This is mainly attributed to the efficient computation of global timeline division (that can be implemented in

a single all-to-all collective communication) and the effectiveness of load balancing based on the global timeline division.

3.5 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. This allows further localized processing on the individual node's aggregation sub-result in a distributed and possibly more efficient manner.

3.6 Greedy Time Division Merge with Centralization (GTDM+C)

This algorithm is another variant of TDM+C, improving the performance by an intelligent global partition assignment policy that attempts to minimize the number of leaves redistributed. For the TDM variants, we assign global partitions to worker nodes in a naive manner. This assignment policy may cause large data movements between worker nodes especially when the partitioning does not match the way the global time divisions are calculated. For illustration, suppose there are one data set, which is partitioned into p workers in two different ways. In the first partitioning, each time division i matches exactly the timeline of the data set of worker i. In the second partitioning, time division i doesn't match the timeline of the data set 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, but large data movement in the second.

This algorithm assigns each timeline partition to a worker node that owns the maximum number of leaves in corresponding partition. The performance gap between TDM and GTDM on the two kinds of partitioning is shown in Figure 9.

3.7 Greedy Time Division Merge (GTDM)

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

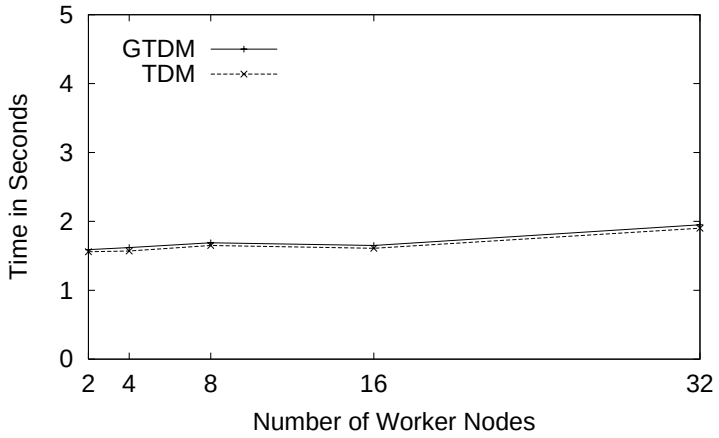
4 Empirical Evaluation

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 [8] was necessary.

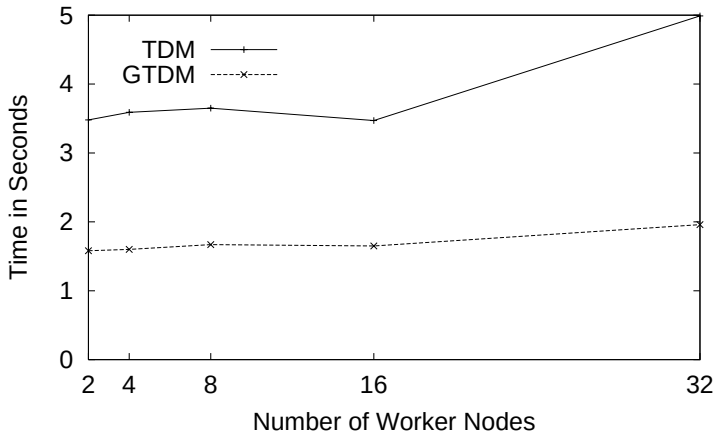
4.1 Experimental Environment

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

Each machine was booted with version 2.0.34 of the Linux kernel. For message passing between the Pentium nodes, we used the LAM implementation of the MPI communication standard [2].



(a) Timeline match



(b) Timeline mismatch

Figure 9: Performance gap between GTDM and TDM

With the LAM implementation, we observed an average communication latency of 790 microseconds and an average transfer rate of about 5 Mbytes/second.

4.2 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 data sets, each tuple had three attributes, an SSN attribute (9 bytes) which was filled with random values, a StartDate attribute (16 bytes), and an EndDate attribute (16 bytes). The SSN attribute refers to an entry in a hypothetic employee relation. The StartDate and EndDate attributes were temporal instants which together construct 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 data sets could have more tuples before their size made them difficult to work with. The total database size for the scale-up experiment at 62 processing nodes was 62 partitions $\times$ 65536 tuples $\times$ 41 bytes = 166 Mbytes.

NumProcessors depends on the type of performance measurement. Scale-up experiments used 2, 4, 8, 16, 32 and 62 processing nodes, while the variable reduction experiments used a fixed set of 32 nodes. Two 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 based on temporal and non-temporal attributes [4].

All experiments except the single speed-up test used a fixed database *partition size* of 65,536 tuples. This was done to facilitate cross-referencing of results between different tests. Because of this, the 32-node results of the scale-up experiments are directly comparable to the results of the 32-node data reduction experiment.

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.

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

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

where $U \geq 2$ is the number of *unique* time stamps in input data set and $A \geq 2$ is the number of all time stamps in input data set. A reduction of 100 percent means that a 100-tuple data set produces 1 leaf in the final aggregation tree because all the tuples have identical StartDates and EndDates. The higher the reduction is, the smaller the size of the aggregation tree is, 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 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.

Parameters	Actual Values
Partitioning	SSN
Number of Processors (p)	2, 4, 8, 16, 32, 62
Tuple Size in bytes	41
Tuples per Processor	65,536
Total Number of Tuples	$p \times 65,536$
Reduction	0 percent

Table 3: Experimental Parameters (Baseline Scale-Up, No Reduction, SSN Partitioning)

4.3 Synthetic Data sets

We set up our first experiment to compare the scale-up properties of the proposed algorithms on a large data set 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.

4.3.1 Baseline Scale-Up Performance: No Reduction/SSN Partitioning

For this experiment, a synthetic data set containing 2M tuples was generated. Each tuple had a randomized SSN attribute and was associated with distinct periods of unit length (i.e., $EndDate = StartDate + 1$). The data set was then 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 EndDate fields.

To measure the scale-up performance of the proposed algorithms, a series of six runs having 2, 4, 8, 16, 32, and 62 nodes, respectively, were carried out. Note that since we fixed the size of the data set on each node, increasing the number of processors meant increasing the total database size. Timing results from this experiment are plotted in Figure 10 and lead us to the following conclusions.

SM performs better than SAT. Intuitively, since the data set 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 and constructing the final results is more efficient than SAT's strategy of having to insert each valid-time period into the final aggregation tree.

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.

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 data set and the fact that scale-up experiments work with more data as the number of nodes increase.

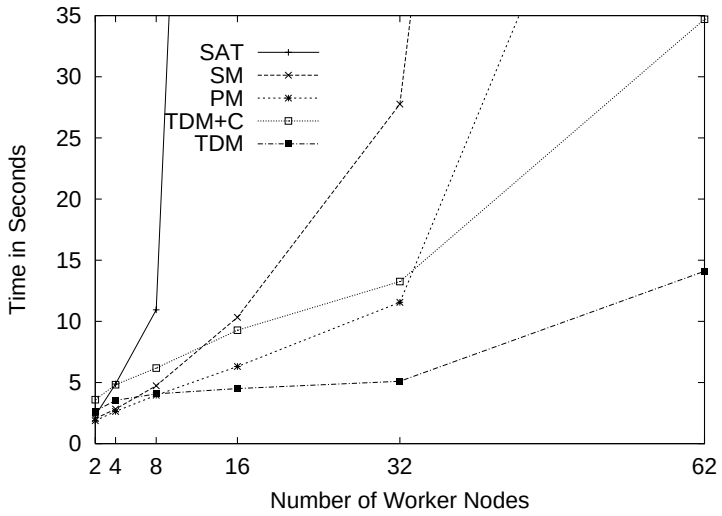


Figure 10: Experimental Results (Baseline Scale-Up, No Reduction, SSN Partitioning)

PM outperforms TDM+C up to 32 nodes. This is attributed to the multiple merge levels needed by PM. A PM computation needs at least $\log_2 p$ merge levels. On the other hand, the TDM+C algorithm only merges local trees once but has three synchronization steps, as described in Section 3 (steps 4, 5 and 6 in Figure 5). 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.

4.3.2 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 4 gives the parameters for this experiment. This experiment was modeled after the first one but with a synthetic data set having 100% (local) reduction. This data set was generated by associating all tuples on each with the same period (for complete local reduction); the SSN attribute values were random.

All algorithms benefit from the 100% data reduction. Comparing results from the baseline experiment with results from the current experiment leads 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.

On the other hand, PM and TDM+C both benefit by the high degree of data reduction enough to make them perform as well as TDM. Because TDM does not transfer local aggregation tree leaves from one node to another, it does not encounter decreased communication costs due to high data reduction. PM and TDM+C, on the other hand, extensively pass tree leaves across processing nodes that communication costs are decreased enough to allow them to perform as well as TDM.

Parameters	Actual Values
Partitioning	SSN
Number of Processors (p)	2, 4, 8, 16, 32
Tuple Size in bytes	41
Tuples per Processor	65,536
Total Number of Tuples	$p \times 65,536$
Reduction	100 percent

Table 4: Experimental Parameters (Scale-Up, 100% Reduction, SSN Partitioning)

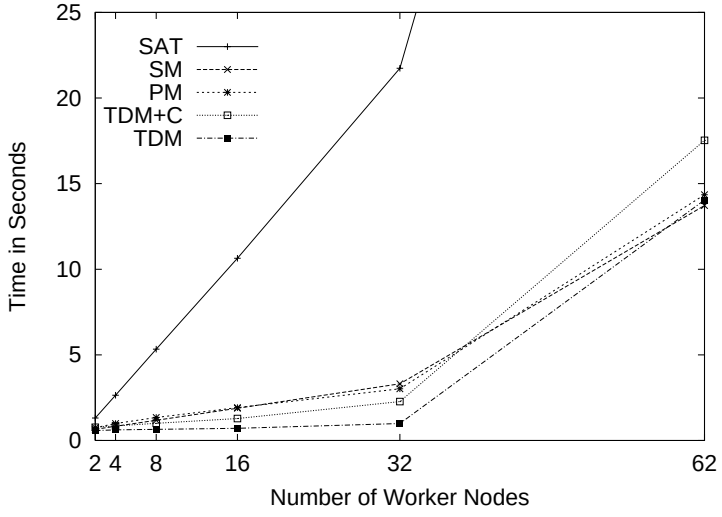


Figure 11: Experimental Results (Scale-Up, 100% Reduction, SSN Partitioning)

Parameters	Actual Values
Partitioning	SSN
Number of Processors (p)	32
Tuple Size in bytes	41
Tuples per Processor	65,536
Total Number of Tuples	$p \times 65,536$
Reduction	0/10/40/60/80/100 percent

Table 5: Experimental Parameters (Scale-Up, Variable Reduction, SSN Partitioning)

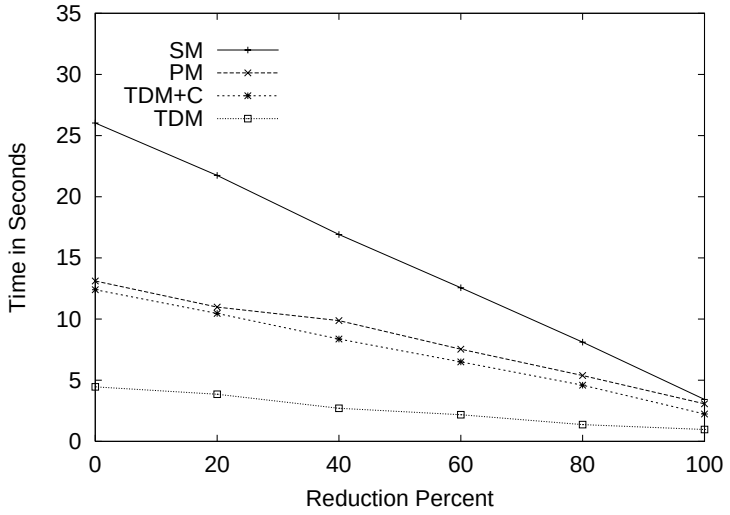


Figure 12: Experimental Results (Scale-Up, Variable Reduction, SSN Partitioning)

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.

4.3.3 Scale-Up Performance: 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 data sets with different reduction were generated. The experiment setting is provided in Table 5 and timing results are plotted on Figure 12. Note that the values plotted for a reduction of 100% correspond to those plotted in Figure 11 for 32 worker nodes and the values plotted for a reduction of 0% correspond to those plotted in Figure 10 for 32 nodes.

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

Parameters	Actual Values
Partitioning	Time
Number of Processors (p)	2, 4, 8, 16, 32, 62
Tuple Size in bytes	41
Tuples per Processor	65,536
Total Number of Tuples	$p \times 65,536$
Reduction	0 percent

Table 6: Experimental Parameters (Scale-Up, No Reduction, Time Partitioning)

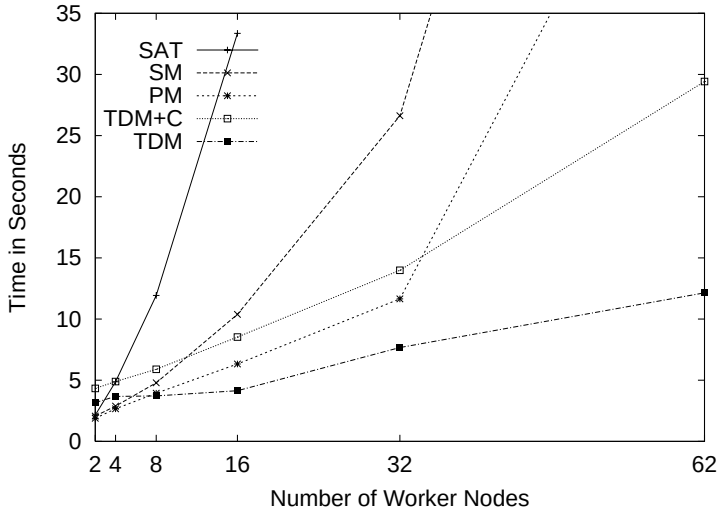


Figure 13: Experimental Results (Scale-Up, No Reduction, Time Partitioning)

4.3.4 Scale-Up Performance: No Reduction/Time Partitioning

This experiment was designed to measure the effect of time partitioning on the scale-up properties of the proposed algorithms. The data set 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 data set by the StartDate attribute and then distributing it to the processing nodes. The experimental settings are summarized in Table 6 and the timing results are provided in Figure 13.

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. In fact, it even performed a little bit poorer in all but the 16-node run (compare with Figure 10). We attribute the small performance gaps to differences in how the partitioning strategies interacting with the number of nodes made TDM redistribute mildly varying numbers of leaves across the runs. Section 4.5 will show how much the greedy assignment policy can improve the performance of the GTDM and GTDM+C algorithms. 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.

Parameters	Actual Values
Partitioning	Time
Number of Processors	32
Tuple Size in bytes	41
Tuples per Processor	65,536
Total Number of Tuples	$32 \times 65,536 = 2,097,152$
Reduction	0/20/40/60/80/100 percent

Table 7: Experimental Parameters (Scale-Up, Variable Reduction, Time Partitioning)

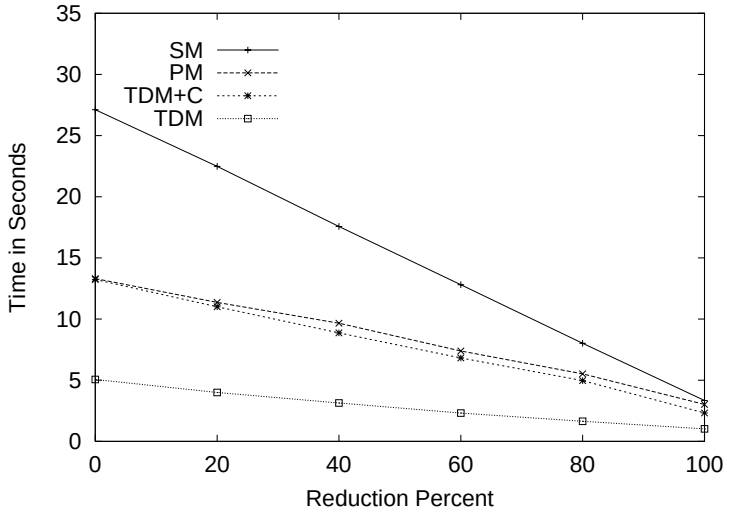


Figure 14: Experimental Results (Scale-Up, Variable Reduction, Time Partitioning)

4.3.5 Scale-Up Performance: 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 7, summarizes the parameters for this experiment. Timing results for this experiment are plotted on Figure 14.

Increasing the amount of data reduction improved the performance of the proposed algorithms. Like the second 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.

Varying data reduction doesn't affect TDM much. The low slope of TDM's performance curve in Figure 14 as compared with Figure 12 shows us that it is the algorithm 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 in the second experiment.

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	$p \times 2,620$
Reduction	80.76 percent

Table 8: Experimental Parameters (Scale-Up, SSN Partitioning)

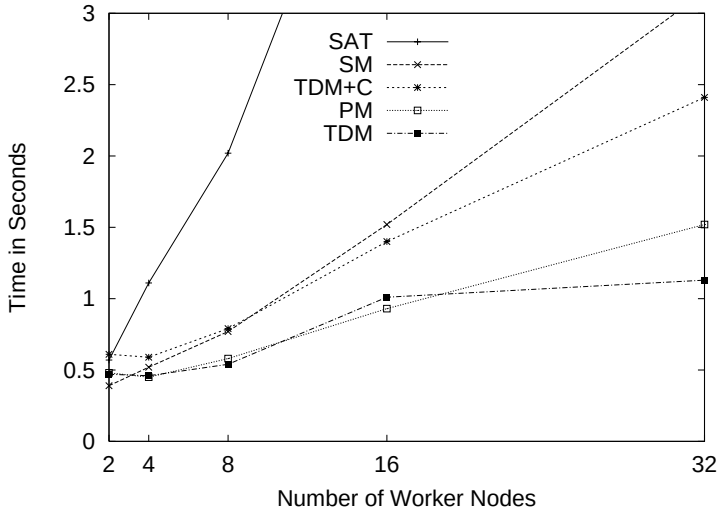


Figure 15: Experimental Results (Scale-Up, SSN Partitioning)

4.4 A Real-World Data set

Tuples in the synthetic data sets 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 data set* [7]. For this data set, the tuple size, and database size were necessarily fixed, at 83,857 tuples and 7.8 Mbytes. For this reason, we used a maximum of 32 processors. Also, the UIS data set used in this experiment has tuples with timelines of variable length.

4.4.1 Scale-Up Performance: SSN Partitioning

This experiment was designed to measure the scale-up properties of the proposed algorithms on the UIS data set partitioned by SSN. The data set was sorted by SSN and distributed to the processing nodes. A varying number of nodes was used, thus applying the aggregate over a varying database size. The experimental parameters for this are shown in Table 8, and the results are plotted in Figure 15.

All the proposed algorithms exhibit similar behavior with the real data set as with synthetic data set. This experiment shows that the relative order of the algorithms applied to the real data set is consistent with the results observed previously (compare the relative positions at 32 nodes with Figure 12 at 80% reduction.)

Parameters	Actual Values
Partitioning	SSN
Number of Processors	2, 4, 8, 16, 32
Tuple Size in bytes	93
Tuples per Processor	41,928/20,964/10,482/5,241/2,620
Total Number of Tuples	83,857
Reduction	97.26/95.26/91.94/87.04/80.51

Table 9: Experimental Parameters (Speed-Up, SSN Partitioning)

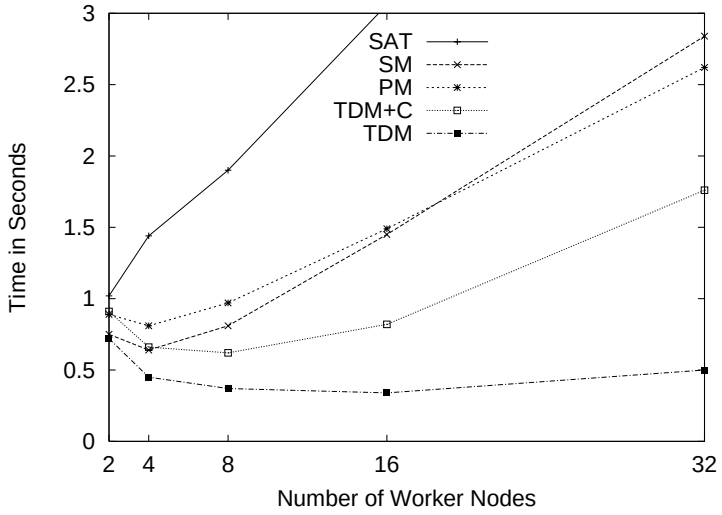


Figure 16: Experimental Results (Speed-Up, SSN Partitioning)

4.4.2 Speed-Up Performance: SSN Partitioning

The data set was sorted by SSN then distributed to a varying number of nodes participating in the experiment. Unlike the scale-up experiment, data size was fixed here. The experimental parameters for this are shown in Table 9. The local reduction falls as fewer tuples are allocated to each processor, because the chance that the period of validity of a tuple will match that of another tuple falls. Timing results from this experiment are plotted in Figure 16.

TDM has better speedup property compared with other algorithms. In fact, the other algorithms actually slowed down as processors were added. Theoretically speaking, as we increase the number of processing nodes, the performance should be improved, because the data size is fixed and processing power increases. However, if there are many synchronizations among the processing nodes, then we may not see noticeable improvement. In order to observe such an improvement, a data set large enough to hide the synchronization overhead is necessary. As we can see in the graph, however, TDM exhibits better speedup property due to its high scalability, but still suffered with more than 16 nodes.

4.4.3 Scale-Up Performance: Time Partitioning

The experimental parameters for this are shown in Table 10, with the results given in Figure 17. We can observe analogous results to the same experiment on data set partitioned by SSN (cf. Fig-

Parameters	Actual Values
Partitioning	Time
Number of Processors (p)	2, 4, 8, 16, 32
Tuple Size in bytes	93
Tuples per Processor	2,620
Total Number of Tuples	$p \times 2,620$
Reduction	80.76 percent

Table 10: Experimental Parameters (Scale-Up, Time Partitioning)

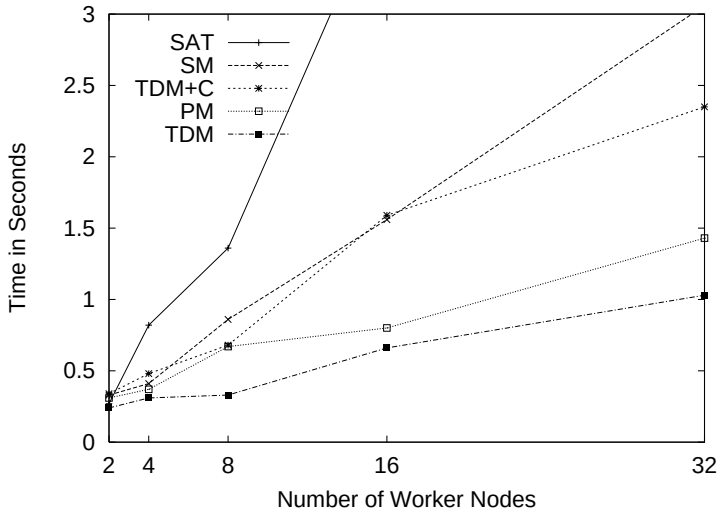


Figure 17: Experimental Results (Scale-Up, Time Partitioning)

Parameters	Actual Values
Partitioning	Time
Number of Processors	2, 4, 8, 16, 32
Tuple Size in bytes	93
Tuples per Processor	41,928/20,964/10,482/5,241/2,620
Total Number of Tuples	83,857
Reduction	97.29/95.32/92.08/87.24/80.76 percent

Table 11: Experimental Parameters (Speed-Up, Time Partitioning)

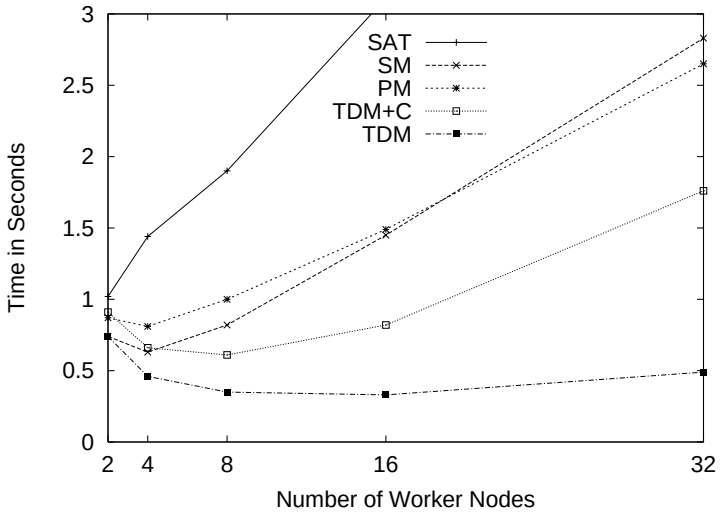


Figure 18: Experimental Results (Speed-Up, Time Partitioning)

ure 15). Just like the results of several experiments on synthetic data sets 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.

4.4.4 Speed-Up Performance: Time Partitioning

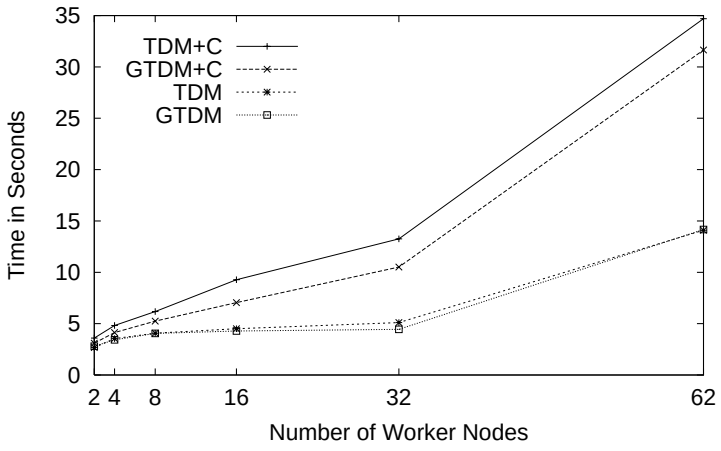
The experimental parameters for this are shown in Table 11 and results from this experiment are plotted in Figure 18. The experiment shows a similar result to that of the same experiment on SSN partitioned data set (cf. Figure 16).

4.5 Performance comparison between GTDM and TDM

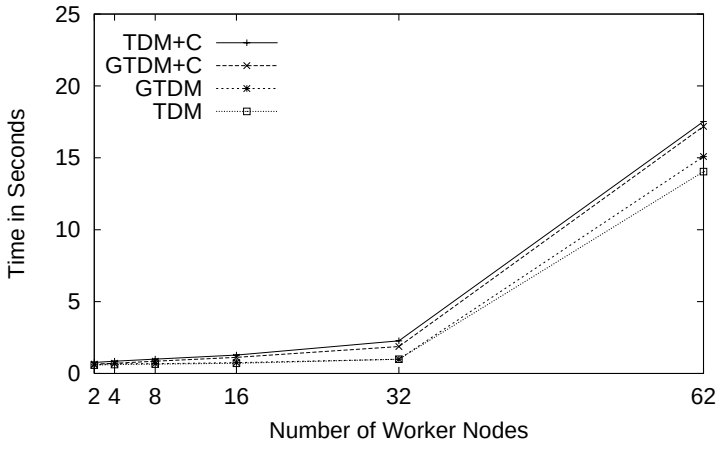
In this section, we compare GTDM and TDM, and their centralized counterparts, by performing the same experiments as those done previously.

4.5.1 Scale-Up Performance: No Reduction/SSN Partitioning/Large Data set

GTDM+C performs slightly better than TDM+C when running on SSN partitioned data. 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 reduce data movements significantly. However, as shown in Figure 19(a), we can observe slight performance improvement even in this case because of minimized network traffic caused by the greedy assignment policy.

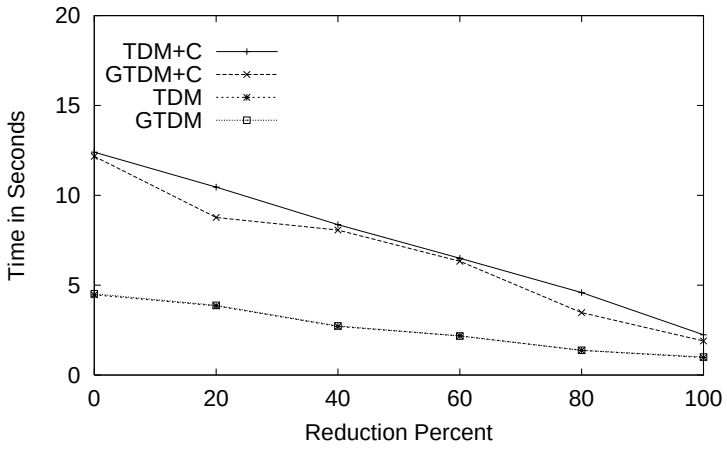


(a) No Reduction, SSN Partitioning, Large dataset



(b) 100% Reduction, SSN Partitioning, Large dataset

Figure 19: Experimental Results (Synthetic Data set with SSN Partitioning)



(b) Variable Reduction, SSN partitioning, Large dataset

Figure 20: Experimental Results (Synthetic Data set with Variable Reduction)

4.5.2 Scale-Up Performance: 100% Reduction/SSN Partitioning/Large Data set

GTDM also benefits from 100% data reduction. Since number of leaves in the aggregation tree in each worker is small because of 100% reduction, GTDM also takes advantage of this (Figure 19(b)).

4.5.3 Scale-Up Performance: Variable Reduction/SSN Partitioning/Large Data set

As the reduction increases, the performance improves. Since the reduction implies the amount of tuples with same time stamp in data sets, as the reduction increases, so does the number of identical tuples (Figure 20).

4.5.4 Scale-Up Performance: No Reduction/Time Partitioning/Large Data set

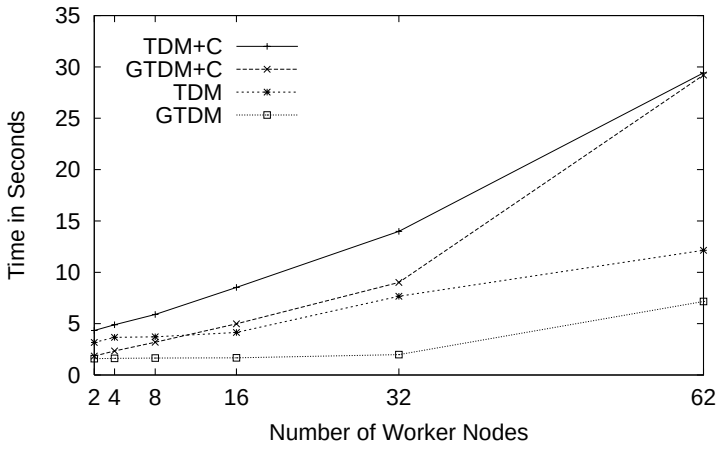
GTDM can outperform TDM when the data set is Time partitioned. The data set used in this experiment is partitioned by time and there is no reduction in it. So, there is no overlaps among timelines of covered by each worker. Since TDM assigned each partition of the data set to each worker in a random manner, the timeline of worker i is not necessarily equal to the time division i of global partition. Consequently, a significant amount of data movements among workers were required in TDM. In GTDM, however, data movements could be minimized because of the greedy assignment policy (Figure 21(a)).

4.5.5 Scale-Up Performance: Variable Reduction/Time Partition/Large Data set

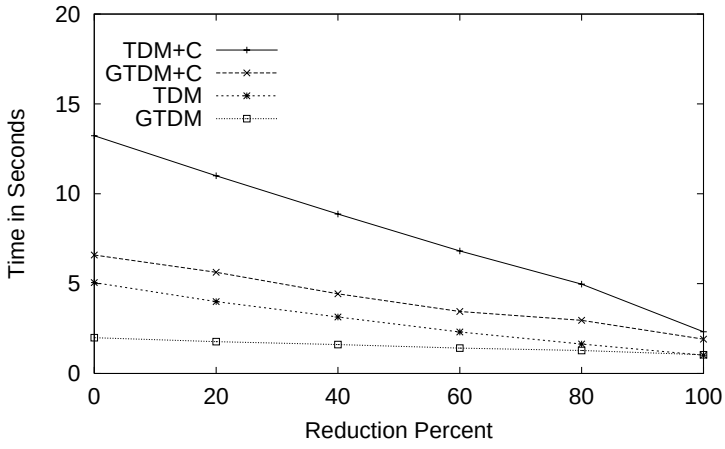
Varying data reduction doesn't significantly affect GTDM. The low slope of GTDM's performance curve in Figure 21(b) shows us that it is the algorithm less affected by variations in local reduction.

4.5.6 Scale-Up Performance: SSN Partitioning/Real World Data set

GTDM outperforms TDM with the real data set. Since it is more probable that data movements among workers happen in real data set than in synthetic data set, GTDM's greedy assignment policy will play an important role in reducing such data movements. GTDM performed better than TDM (Figure 22(a)).

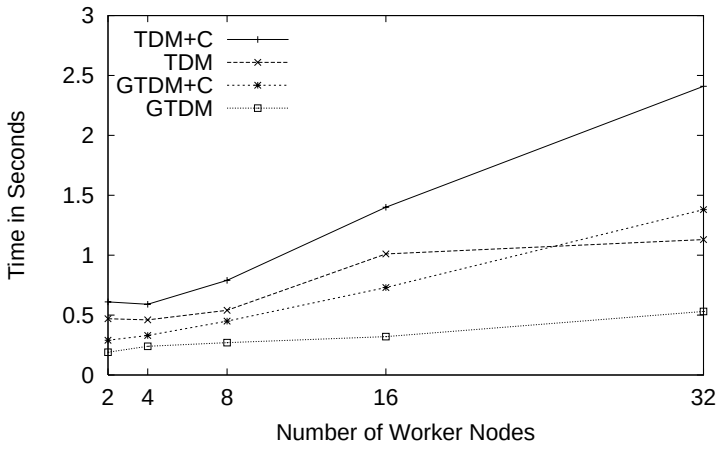


(a) No Reduction, Time Partitioning, Large dataset

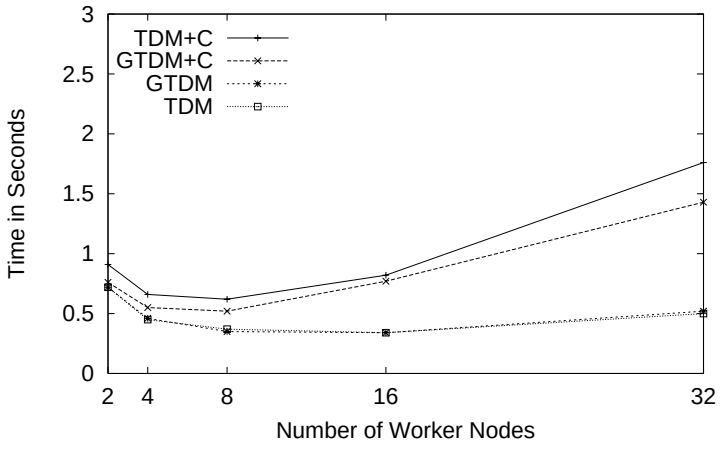


(b) Variable Reduction, Time partitioning, Large dataset

Figure 21: Experimental Results (Synthetic Data set with Time Partitioning)

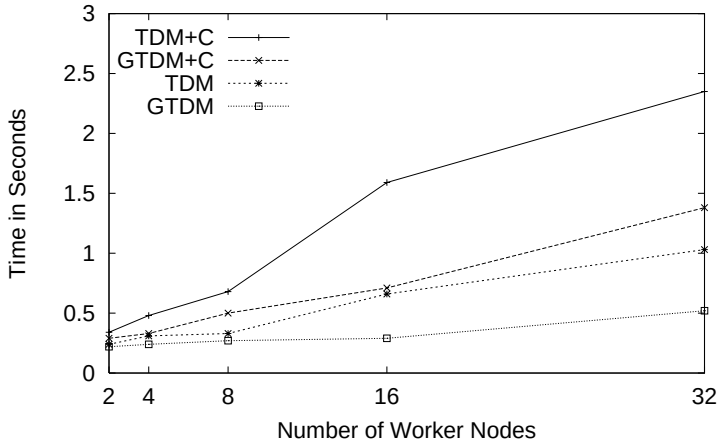


(a) Scale-up, SSN Partitioning, Small dataset

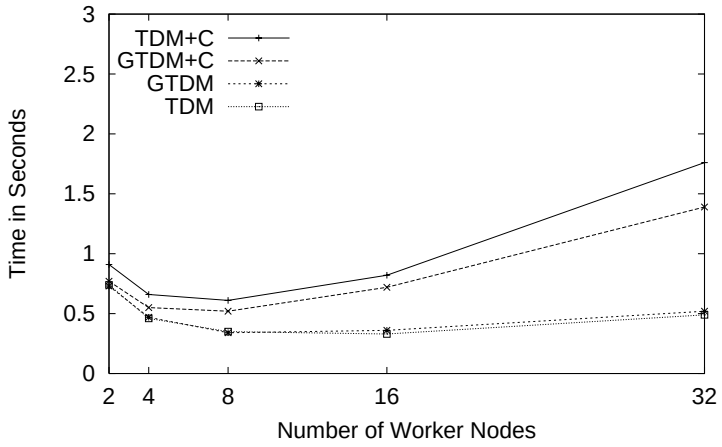


(b) Speed-Up, SSN Partitioning, Small dataset

Figure 22: Experimental Results (Real Data set with SSN Partitioning)



(a) Scale-Up, Time Partitioning, Small dataset



(b) Speed-Up, Time Partitioning, Small dataset

Figure 23: Experimental Results (Real Data set with Time Partitioning)

4.5.7 Speed-Up Performance: SSN Partitioning/Real World Data set

TDM and GTDM show similar speedup property with the real data set. GTDM also has better speedup property compared with other proposed algorithms. However, since we used a relatively small-sized data set, we couldn't see any big difference in speedup between GTDM and TDM (Figure 22(b)).

4.5.8 Scale-Up Performance: Time Partitioning/Real World Data set

GTDM outperforms TDM with the real data set. As with the Time partitioned synthetic data set, GTDM exhibits substantial improvement over TDM for the Time partitioned real world data set, due to minimized data movements.

Data Reduction	Node Count	Distributed Results	Centralized Results
HI	Small	**GTDM**	**GTDM+C**
	Large	**GTDM**	**PM**
LOW	Small	**GTDM**	**PM**
	Large	**GTDM**	**GTDM+C**

Table 12: Matrix of Recommendations

4.5.9 Speed-Up Performance: Time Partitioning/Real World Data set

The experiment (Figure 23(b)) shows similar results to that of the same experiment on the SSN partitioned data set (cf. Figure 16).

4.6 Summary

The empirical observations confirm that data set partitioning, result placement, data reduction effected by the aggregation, and the number of processing nodes all affect the proposed algorithms, in different ways. SAT and SM, as seen in Figures 10, 11, and 13, were affected most by the number of processing nodes. Figure 14 shows that SM, SAT, PM and TDM+C were significantly slowed by low data reduction while TDM was the least affected. Also, Figures 10, 11, and 13 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 11) and low reduction and small configurations (Figures 10 and 13). Data set partitioning only affected the TDM variants, and even then, not substantially (compare Figure 10 with Figure 6). These results parallel those for the real-world data set (Figures 15–18).

GTDM (and GTDM+C) has virtually the same performance as TDM (and TDM+C) for SSN partitioning (Figure 19). For time partitioning, GTDM differs from TDM only with low reduction. GTDM+C differs from TDM+C only with low reduction *and* small to medium configurations. In all cases, the greedy variant was superior. GTDM was not sensitive to the data partitioning.

5 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. 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 12.

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 and a small configuration.

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. In the few cases where this was less effective than PM, the difference was generally less than 10%.

Our experimental observations lead us to the following issues for future research.

1. *Impact of skew.* In a temporal aggregate query with tuple placement and/or selection skew, some worker nodes will complete its 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 [16]. However, the specific impact of skew should be investigated.

2. *Load balancing.* Uneven computing time on the processing nodes as caused by data set 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.

3. *Disk-paging strategies.* 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 [8] would allow the algorithms to handle larger data set sizes.

4. *Deeper sensitivity analysis to other factors.* 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.

5. *Impact of grouping.* 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*."

References

[1] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[2] Ohio Supercomputer Center. LAM/MPI parallel computing. http://www.osc.edu/lam.html, 1998.

[3] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I. Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[4] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[5] Robert Epstein. Techniques for processing of aggregates in relational database systems. Technical Report UCB/ERL M7918, University of California, Berkeley, CA, February 1979.

[6] Johann C. Freytag and Nathan Goodman. Translating aggregate queries into iterative programs. In *Proceedings of the 12th VLDB Conference*, pages 138–146, Kyoto, Japan, 1986.

[7] Jose A. G. Gendrano, Rachana Shah, Richard T. Snodgrass, and Jian Yang. University information system (uis) dataset. Technical Report TIMECENTER CD-1, Department of Computer Science, University of Arizona, September 1998.

[8] Nick Kline. *Aggregation in Temporal Databases.* PhD thesis, Computer Science Department, University of Arizona, May 1999.

[9] Nick Kline and Richard T. Snodgrass. Computing temporal aggregates. In *the 11th Inter. Conference on Data Engineering*, pages 222–231, Taipei, Taiwan, March 1995.

[10] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the ACM SIGMOD International Conference*, pages 28–36, 1988.

[11] Richard T. Snodgrass, Santiago Gomez, and Edwin McKenzie. Aggregates in the temporal query language tquel. *tkde*, 5:826–842, oct 1993.

[12] Michael Stonebraker. The case for shared nothing. *A Quarterly bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 9(1):4–9, March 1986.

[13] Transaction Processing Performance Council (TPC). *TPC Benchmark D (Decision Support), Standard Specification, Revision 1.3.1*, August 1998.

[14] Paul A. Tuma. Implementing historical aggregates in TempIS, 1992. Master's Thesis.

[15] Carolyn Turbyfill, Cyril Orji, and Dina Bitton. As^{3}ap: An ansi sql standard scaleable and portable benchmark for relational database systems. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 4, pages 167–207. Morgan Kaufmann Publishers, 1991.

[16] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th VLDB Conference*, pages 537–548, Barcelona, Spain, September 1991.

[17] Xinfeng Ye and John A. Keane. Processing temporal aggregates in parallel. In *IEEE Internation Conference on Systems, Man, and Cybernetics*, pages 1373–1378, Orlando, FL, October 1997.