Parallel Algorithms for Computing Temporal Aggregates *

Jose Alvin G. Gendrano[†] *Bruce C. Huang*[‡] *Jim M. Rodrigue*[§]
Bongki Moon[†] *Richard T. Snodgrass*[†]

[†]Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
{jag,bkmoon,rts}@cs.arizona.edu

[‡]IBM Storage Systems Division
9000 S. Rita Road
Tucson, AZ 85744
brucelee@us.ibm.com

[§]Raytheon Missile Systems Co.
1151 East Hermans Road
Tucson, AZ 85706
jmrodrigue@west.raytheon.com

Technical Report 98-9

Abstract

The ability to model the temporal dimension is essential to many applications. Furthermore, the rate of increase in database size and 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 a shared-nothing architecture 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 in different ways. We designed the Time Division Merge algorithm to produce distributed result placement, as differentiated from the centralized result strategies used by the other proposed algorithms. For centralized results and high data reduction, we found that the Pairwise Merge algorithm was preferred regardless of the number of processing nodes, but for low data reduction it was only preferred up to 32 nodes, while a variant of Time Division Merge was best for larger configurations.

September, 1998

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

*This work was sponsored in part by National Science Foundation grants CDA-9500991 and IRI-9632569.

1 Introduction

Aggregate functions are an essential component of data query languages, and are heavily used in many applications such as data warehousing. Hence, efficient execution of aggregate functions is an important goal. Aggregate computation is 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. For example, finding the (time-varying) maximum salary of professors in the Computer Science Department involves computing the temporal extent of each maximum value, which requires determining the tuples that overlap each temporal instant.

The appeal of parallel processing technology becomes even stronger as the size of many data-intensive applications become large, as can be seen in OLAP systems and data warehousing environments [3, 4]. Although there exist efficient algorithms for computing temporal aggregates [9, 10, 12], to the best of our knowledge, no parallel algorithm has been developed for computing temporal aggregates.

In this paper, we present several new parallel algorithms for the computation of temporal aggregates on a shared-nothing architecture [11]. Specifically, we focus on the Aggregation Tree algorithm [9] and propose several approaches to parallelize that algorithm. The performance of the parallel algorithms relative to various data set and operational characteristics is our main interest.

The rest of 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. Section 4 presents empirical results obtained from the experiments performed on a shared-nothing Pentium cluster. Finally, section 5 concludes the paper and gives an outlook to future work.

2 Background and Related Work

There are two types of non-temporal aggregates in relational database systems, scalar aggregates and aggregate functions [7]. Scalar aggregates are operations such as average, min, max, count, and sum that produce a single value over an entire relation, while aggregate functions first partition a relation based on some attribute value and then compute scalar aggregates independently on the individual partitions.

Simple algorithms for evaluating scalar aggregates and aggregate functions were discussed by Epstein [7]. A different approach employing program transformation methods to systematically generate efficient iterative programs for aggregate queries has also been suggested [8]. Tumas extended Epstein's algorithms to apply to temporal aggregates [12]; these were further extended by Kline [9]. While the resulting algorithms were quite effective in a uniprocessor environment, all suffer from poor scale-up performance, which identifies the need to develop parallel algorithms to compute 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 more recent Gamma database machine project [6] implemented similar scalar aggregates and aggregate functions on a shared-nothing 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 following example [9], there are 4 tuples to be inserted into an empty aggregation tree (see Table 1(a)). 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 in 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

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

(a) Input Database Records

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

(b) Temporal Aggregation Results

Table 1: Sample Database and Its Temporal Aggregation

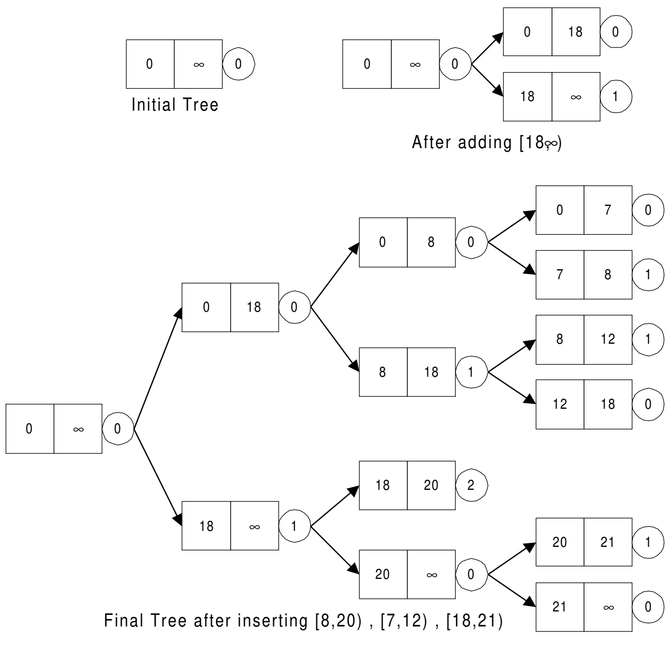

Figure 1: Example run of the Sequential(SEQ) Aggregation Tree Algorithm

parents' counts is $0+0+1=1$ and adding the leaf count, gives a total of 2. The temporal aggregate results are given in Table 1(b).

It should be noted that the order of tuple insertion into the aggregation tree affects its performance, thought not its 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.

Though SEQ correctly computes temporal aggregates, it is still a sequential algorithm, bounded by the resources of a single processor machine (see Figure 2(a)). Therefore, a parallel method for computing temporal aggregates is needed.

3 Parallel Processing of Temporal Aggregates

In this section, we propose five parallel algorithms for the computation of temporal aggregates. We start off with two simple parallel extensions to the sequential Aggregation Tree algorithm, the Single Aggregation Tree (abbreviated SAT) and Single Merge (SM) algorithms. We then go on to introduce the Time Division Merge with Centralizing step (TDM+C) and Pairwise Merge (PM) algorithms, which hopefully will perform better, albeit requiring more coordination. Finally, we present the Time Division Merge (TDM) algorithm, a variant of TDM+C, which distributes the resulting relation, as differentiated from the centralized results produced by the other algorithms.

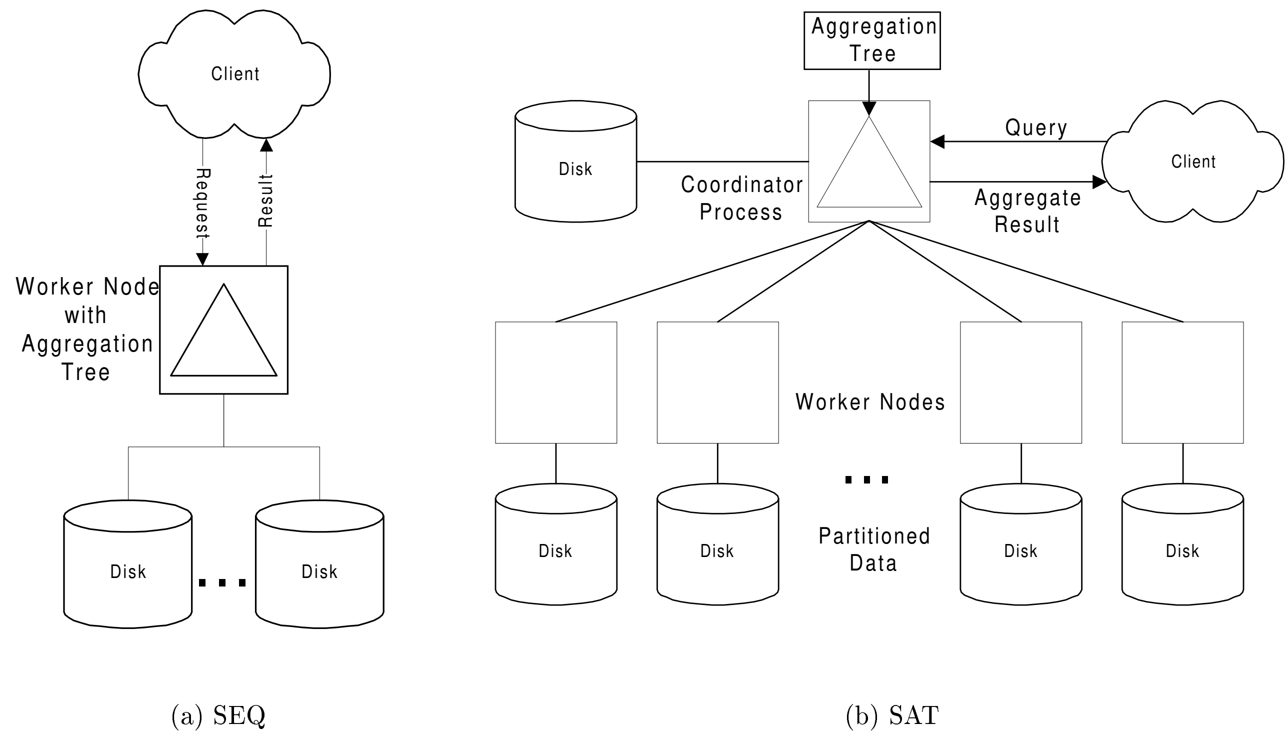

Figure 2: SEQ and SAT Algorithms

3.1 Single Aggregation Tree (SAT)

The first algorithm, SAT (see Figure 2(b)), 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, and sends these periods up to the coordinator, along with the attribute over which the aggregate is being computed (except for the `COUNT` aggregate). The central coordinator receives the periods from all the worker nodes, builds the complete aggregation tree, and returns the final result to the client.

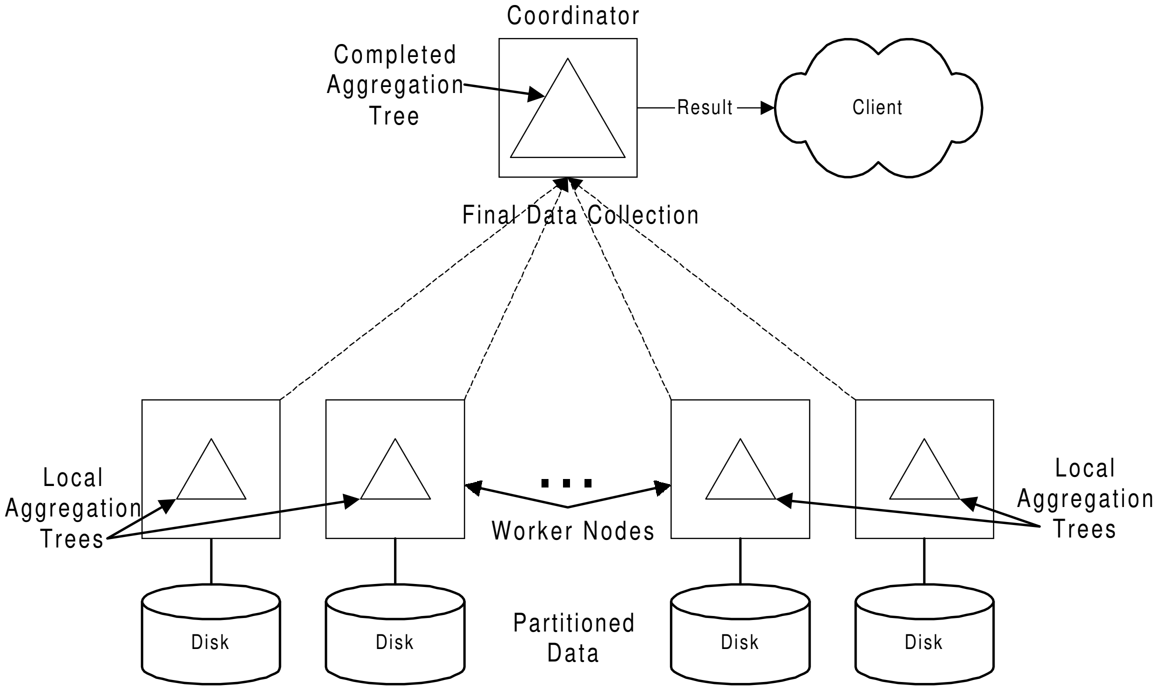

Figure 3: Single Merge (SM) Algorithm

3.2 Single Merge (SM)

The second parallel algorithm, SM (see Figure 3), 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, based on its data partition. Upon finishing their local aggregation trees, the worker nodes send the coordinator a `Done` message. The coordinator keeps track of the worker nodes who have finished, and polls them sequentially for their aggregation tree's leaves. For increased parallelism, the coordinator does not wait until all worker nodes finish and starts polling for leaves as soon as the first `Done` message is received.

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. 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.

3.3 Time Division Merge with Coordinator (TDM+C)

Like SM, the third parallel algorithm also extends the aggregation tree method by employing both computational and I/O parallelism (see Figure 4). The main steps for this algorithm are outlined in Figure 5.

3.3.1 Overall Algorithm

TDM+C starts when the coordinator receives a temporal aggregate request from a client. Each worker node is then instructed to build a local aggregation tree based on its data partition with the knowledge of the number of worker nodes, p, participating in the query.

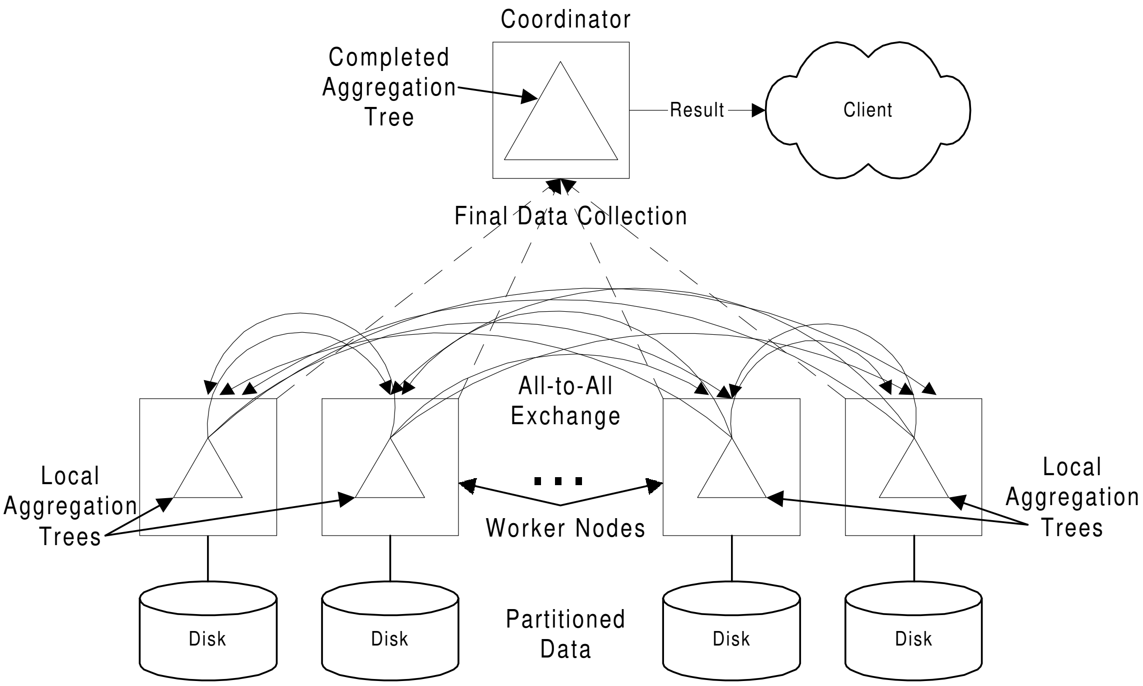

Figure 4: Time Division Merge with Centralizing Step (TDM+C) Algorithm

After each worker node constructs its local aggregation tree, the tree is augmented in the following manner. The node 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 period they represent, and any parent nodes are discarded. After all worker nodes complete their aggregation trees, they exchange minimum (earliest) start time and maximum (latest) end time values to ascertain the overall timeline of the query.

Each local aggregation tree is then split into p local partitions, each consisting of a period and a tuple count, where each partition contains approximately the same number of tuples. In splitting the local aggregation trees, we try to exploit processor parallelism and implement load balancing. Note that this strategy allows the periods of the local partitions to have different durations. The local partition set (containing p partitions) from each processing node is then sent to the central coordinator.

The central coordinator then takes all p local partition sets[1] and computes the p global partitions, one for each worker node (how this is done is discussed in detail in the next section).

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 and respective assignments 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 to. Note that inserted periods that span the periods of more than one partition are cut, and the fragments are sent to the worker nodes assigned to the overlapped partitions (cutting the inserted periods does not affect the correctness of the result).

Worker nodes then merge the leaves they receive with the leaves they already have, and are responsible for computing the temporal aggregate results for the period of the partition assigned to them. When all the worker nodes are done merging, the coordinator polls them for final answers sequentially, in order of the periods of their assigned partitions. Finally, the coordinator merely concatenates the results and returns them to the client. The results are guaranteed not to overlap.

[1] A total of p^2 local partitions are created by p worker nodes.

1. Client request
2. Build local aggregation trees
3. Calculate local partition sets
4. Calculate global partition set
5. Exchange data and merge
6. Merge local results
7. Return results to client

Figure 5: Major Steps for the TDM+C Algorithm

Count	Begin	End
50	5	9
50	9	800
50	800	1500
15	0	30
15	30	350
15	350	10000
30	0	10
30	30	1000
30	1000	5000

Table 2: Local Time Partitions from Three Worker Nodes

3.3.2 Calculating the Global Partition Set

We now examine in more detail the computation of the global partition set by the central coordinator. Recall that the coordinator receives from each processing node a local partition set, consisting of p contiguous partitions, each containing a period and 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 (based on the leaf node information sent by the worker nodes).

As an example, Figure 7 presents 3 local partitions from 3 worker nodes. The number between each hash mark segmenting the local timeline represents the number of leaf nodes intersecting that local partition. The total number of leaf nodes from the 3 nodes is $50 \cdot 3 + 15 \cdot 5 + 30 \cdot 3 = 285$. So the best plan is to have $\frac{285}{3} = 95$ leaf nodes to be processed on each node. Figure 6 illustrates the computation of the global partition set.

We modified the Aggregation Tree algorithm to compute the global partition set, based on the leaf node information sent by the worker nodes. In doing so, we can treat the local partition sets from the participating worker nodes as periods (see Table 2), inserting these periods into the modified aggregation tree. Note that 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 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, instead of just by 1. Second, a parent node cannot have a count value greater than 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's count becomes 0. Third, during an insertion traversal for a

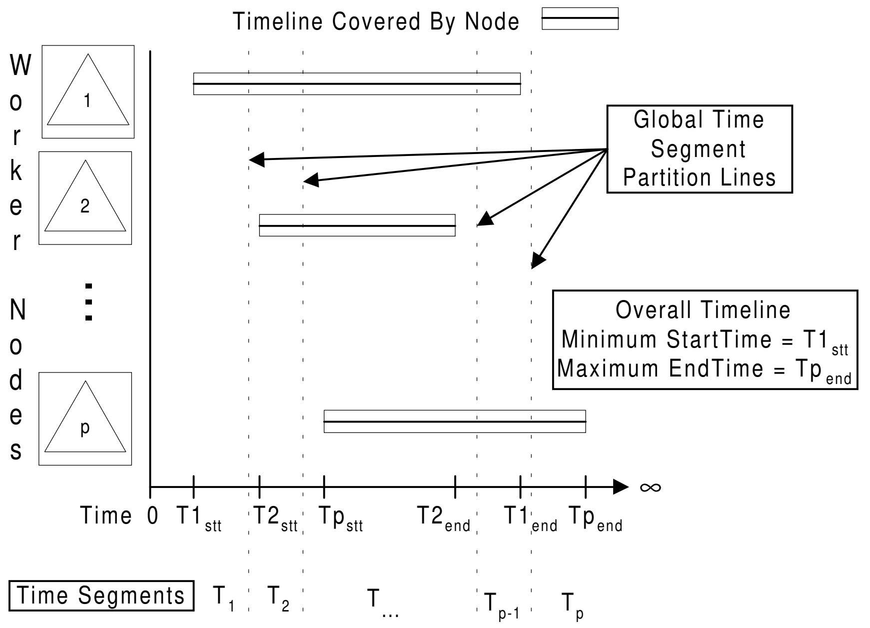

Figure 6: Timeline divided into p partitions, forming a global partition set

record, if the search traversal needs to be split, the record count is split proportionally between the left and right 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 8. 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's count of 30 in proportion to the durations of the left and right sub-trees. The left sub-tree of the root contains a period [0,5) for a duration of 5 time units. The fourth local partition's 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

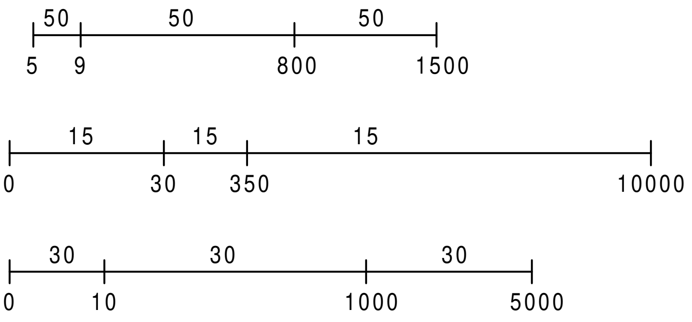

Figure 7: Local Partition Sets from Three Worker Nodes

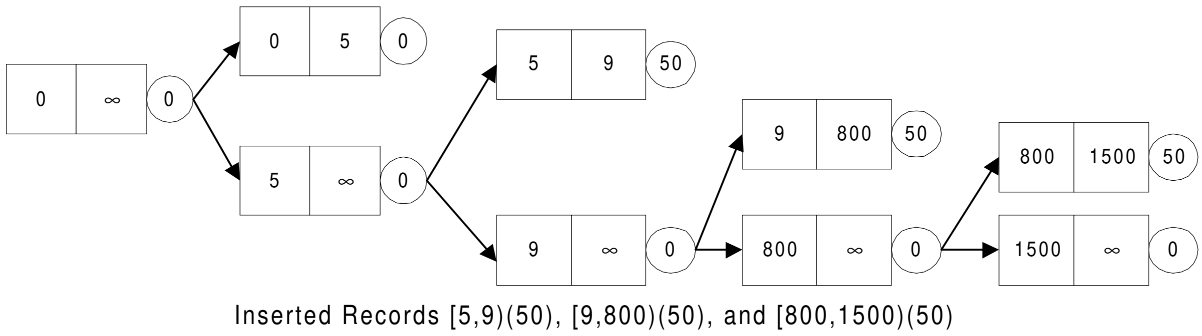

Figure 8: Result of Adding the First 3 Local Time Partitions into the Modified Aggregation Tree

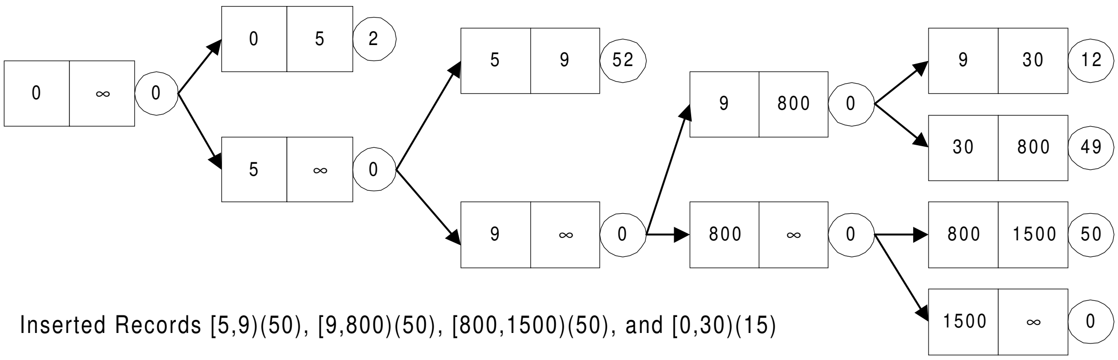

Figure 9: Resulting aggregation tree from adding partition 4 from Table 2 to Figure 8

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 9). We now pass 13 down the right sub-tree, increasing leaf node [5,9)(50) to [5,9)(52) as its share of the newly added partition's count, 2, is added to it. 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 required to be split from [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 $\frac{(30-9)}{(800-9)} \cdot 50 = 1$ 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 9 for the result. Table 3 shows the leaf node values in a tabular format once all 9 local time partitions from Table 2 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. The idea is simple. 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. If the count is more than the optimal number, then we break up the leaf node that causes this node count to be greater than the optimal number. Once again, the leaf node division of counts is done proportionally to the duration of the periods.

As an example, take the output from Table 3. We know the optimal number of periods per global partition is 95. So we add the leaf node counts until we reach node [10,30)(12). The sum at this point is 96,

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

Table 3: All leaf node values in a tabular format once all 9 partitions from Table 2 are inserted

or 1 more than optimal. We break up [10,30)(12) into two leaf nodes such that the first leaf node's period should contain 11 leaf nodes, and the newly created leaf node should contain only 1. Again 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 [0,28), [28,866), and [866,10000).

The reader should be aware that this global time partition resolution algorithm is not perfect. That is, 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. When a local partition says it has 50 leaf nodes in period [9,800), the global partition scheme assumes it is a uniform distribution, even though the actual leaf nodes may be heavily skewed. We lose locality by exchanging local partition information. If we wanted perfect global partitioning, then all the leaf nodes from all the worker nodes must be sent to the coordinator for resolving the time partitions. But that approach imposes too much of an overhead.

3.3.3 Expected Performance

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 three global synchronization steps that may limit the performance obtained. First, all of the local partition sets must be completed before the global time segment partitioning can begin. Second, all of the worker nodes must complete their merges before the client receives its final result from the coordinator process. Third, all the worker nodes must finish sending their final results to the coordinator.

The next algorithm, PM, will attempt to obtain better performance, by replacing the three global synchronization steps with $\log_2 p$ localized synchronization steps.

3.4 Pairwise Merge (PM)

The fourth parallel algorithm, PM (see Figure 10), differs from TDM+C in two ways. First, the coordinator is more involved than in TDM+C. Secondly, instead of all the worker nodes merging simultaneously as in TDM+C, pairs of worker nodes merge when the opportunity presents itself. Which two worker nodes are paired is determined dynamically by the query coordinator.

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. 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.

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

Figure 10: Major Steps for the PM Algorithm

Once a worker node finishes transmitting leaves to its buddy worker node, it is no longer a participant in the query, while the other, now idled, buddy worker node is ready for another merge. This buddying-up continues until the query coordinator can ascertain that only one worker node is left, containing the completed aggregation tree. The query coordinator then directs the sole remaining worker node to submit the results directly to the client. Figure 11 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.

3.5 Time Division Merge (TDM)

The fifth parallel algorithm, TDM (see Figure 12), 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 a much larger distributed query. This allows further localized processing on the individual node's aggregation sub-result in a distributed and possibly more efficient manner.

4 Empirical Evaluation

For the purposes of our evaluation, we chose the temporal aggregate operation `COUNT`. The `COUNT` operation does not require that the attribute itself be sent, thereby simplifying the data structures maintained while still exhibiting the characteristics of a temporal aggregate computation. Based on this temporal aggregate operation we perform a variety of performance evaluations on the five parallel algorithms presented in this paper. The matrix in Table 4 summarizes the experiments we have done.

thereby reducing the communication somewhat (as we will see, communication costs dominate).

Expt. Number	Algorithms Covered	NumProcessors
1	SAT, PM, SM, TDM, TDM+C	2, 4, 8, 16, 32, 64
2	SAT, PM, SM, TDM, TDM+C	2, 4, 8, 16, 32, 64
3	SAT, PM, SM, TDM, TDM+C	2, 4, 8, 16, 32, 64
4	PM, SM, TDM, TDM+C	16

Table 4: Experimental Case Matrix Summary

4.1 Settings for the Experimental Environment

The experiments were conducted on a 64-node shared-nothing cluster of 200MHz Pentium machines, each with 128MB of main memory and 2GB hard disks. The machines were physically mounted on two racks of 32

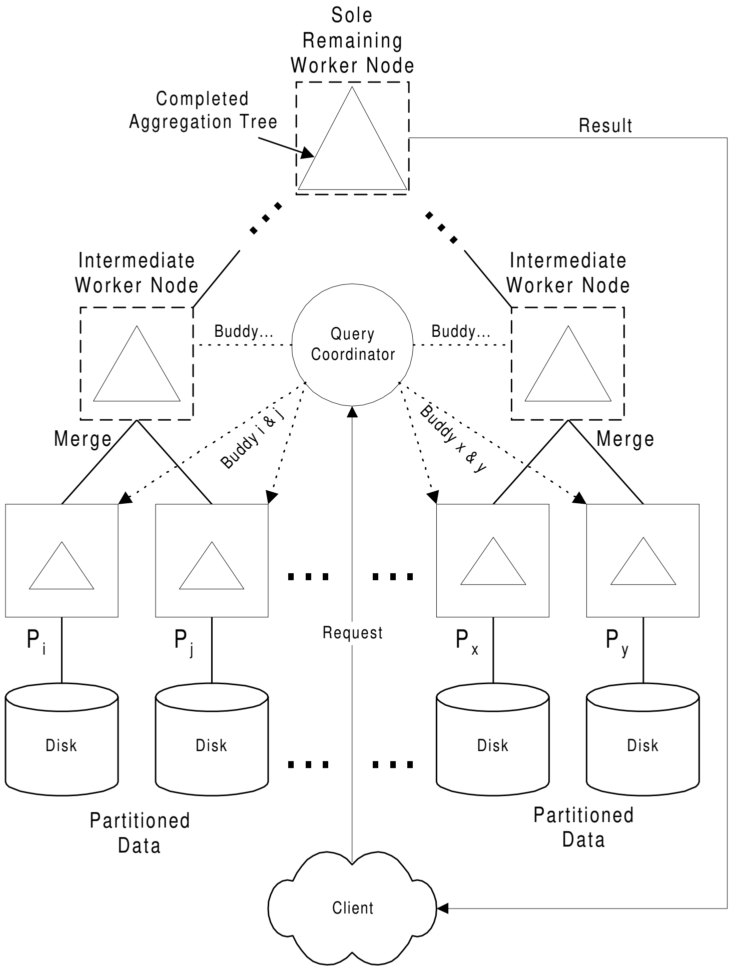

Figure 11: Pairwise Merge (PM) Algorithm

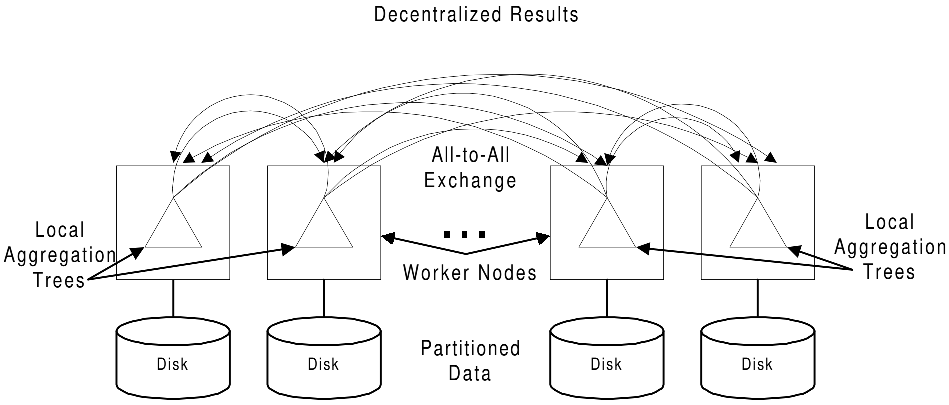

Figure 12: Time Division Merge (TDM) Algorithm

Parameter	Description	Values
$NumProcessors$	Number of processing nodes used	2 to 64
$Partitioning$	How the dataset is partitioned	by StartDate, by SSN
$TupleSize$	Size (in bytes) of each tuple	41 bytes
$PartitionSize$	Number of tuples per partition	65536 tuples
$NumTuples$	Total database size	$NumProcessors \cdot PartitionSize$
$DataReduction$	Percentage decrease of result	0/20/40/60/80/100 percent

Table 5: Experiment Parameters

machines each. 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 2.0.30 of the Linux kernel. For message passing between the Pentium nodes, we used the LAM implementation of the MPI communication standard [2]. 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

In order to help precisely define the parameters for each set of tests, we describe an experiment classification scheme. We later present the experiments in terms of this scheme. Table 5 lists the different parameters, a short description of each, and the set of values used.

Synthetic datasets were generated to model relations which store time-varying information for each employee in a database. Each tuple has three attributes, a hypothetic SSN attribute which is filled with random digits, a StartDate attribute, and an EndDate attribute. The SSN attribute refers to an entry in a hypothetic employee relation. On the other hand, the StartDate and EndDate attributes are temporal instants which together construct a valid-time period. The actual mode of generation varies from one experiment to another and is described later.

NumProcessors depends on the type of performance measurement. Scale-up experiments used 2, 4, 8, 16, 32, and 64 processing nodes, while the variable reduction experiment used a fixed set of 16 nodes.

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 were attempts to model range partitioning based on temporal and non-temporal attributes in a parallel database system [5].

The *tuple size* was fixed at 41 bytes/tuple. This parameter is given, for figuring out the actual size of the dataset, in bytes. Note that 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.[2]

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 16-node results of the scale-up experiments are directly comparable to the results of the 16-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 dataset. A reduction of 100 percent means that a 100-tuple dataset produces 1 leaf in the final aggregation tree because all the tuples have identical StartDates and EndDates. As a general rule, the percentage data reduction is inversely proportional to the percentage of unique StartDates and EndDates across the entire dataset.

[2]The total database size for the scale-up experiment at 64 processing nodes was 64 partitions · 65536 tuples · 41 bytes = 171,966,464 bytes.

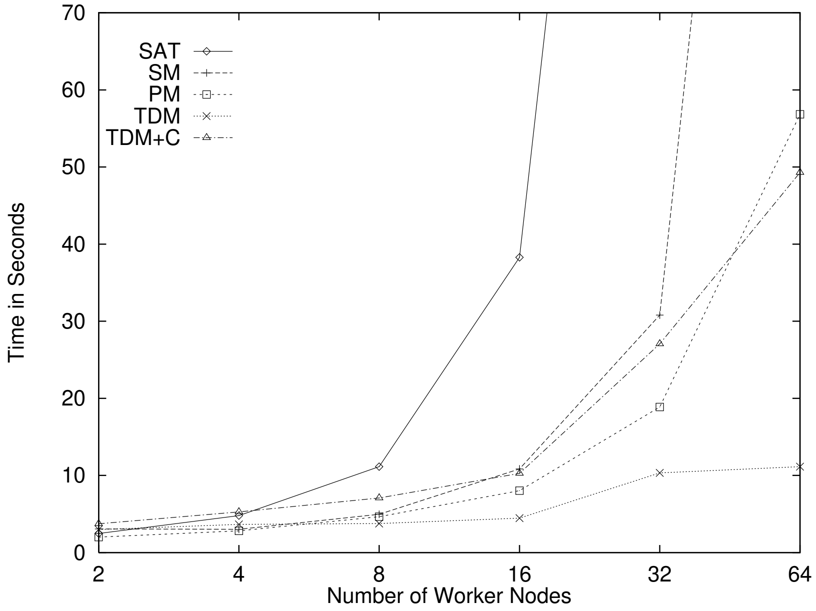

Figure 13: Scale-Up Results (4M tuple Dataset with No Reduction and SSN Partitioning)

4.3 Baseline Scale-Up Performance : No Reduction and SSN Partitioning

We set up our first experiment to compare the scale-up properties of the proposed algorithms on a dataset with no reduction. We will also use the measurements taken from this experiment as a baseline for later comparisons in the subsequent experiments. Table 6 gives the parameters for this particular experiment.

Parameter	**Actual Value**
$NumProcessors$	2, 4, 8, 16, 32, 64
$Partitioning$	by SSN
$TupleSize$	41 bytes
$PartitionSize$	65536 tuples
$NumTuples$	(2,4,8,16,32,64)*65536
$DataReduction$	0 percent

Table 6: Parameters for the No-Reduction with SSN Partitioning Scale-Up Experiment

For this experiment, a synthetic dataset containing 4M tuples was generated. Each tuple had a randomized SSN atrribute and was associated with distinct periods of unit length (i.e., $EndDate = StartDate + 1$). The dataset was then sorted by SSN.[3] These partitions were then distributed to the 64 processing nodes.

To measure the scale-up performance of the proposed algorithms, a series of 6 runs having 2, 4, 8, 16, 32, and 64 nodes, respectively, were 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. Timing results from this experiment are plotted in Figure 13 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.

[3]Since the SSN fields are generated randomly, this has the effect of randomizing the tuples in terms of StartDate and EndDate fields.

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. This sorting makes compiling the results more efficient[4] 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.[5]

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. Because of this additional overhead, TDM will always perform better than TDM+C. 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.

Among the algorithms that provide monolithic results, PM has the best scaleup performance 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 where p is the number of processing nodes. On the other hand, the TDM+C algorithm only merges local trees once but has three synchronization steps, as described in Section 3. 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 suprised to realize that PM was still performing better than TDM+C up to perhaps 50 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.

4.4 Scale-Up Performance : 100% Reduction and SSN Partitioning

This experiment is 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 7 gives the parameters for this experiment.

Parameter	Actual Value
$NumProcessors$	2, 4, 8, 16, 32, 64
$Partitioning$	by SSN
$TupleSize$	41 bytes
$PartitionSize$	65536 tuples
$NumTuples$	(2,4,8,16,32,64)*65536
$DataReduction$	100 percent

Table 7: Parameters for the 100% Reduction with SSN Partitioning Scale-Up Experiment

This experiment is modeled after the first one but with a synthetic dataset having 100% reduction. This dataset was generated by creating 4M tuples associated with the same period and having randomized SSN attributes. The synthetic dataset was then rearranged randomly[6] and split into 64 partitions each having 65,536 tuples.

This experiment, like the first one, is a scale-up experiment. Hence, it was conducted in much the same way. Timing results from this experiment are plotted in Figure 14 and leads us to the following observations.

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.

[4]The SM coordinator uses a merge-sort variant in compiling and constructing the final results.

[5]In SAT, all the periods are sent to the coordinator which builds a single, but large, aggregation tree.

[6]The aggregation tree algorithm performs at its worst case when the dataset is sorted by time [9].

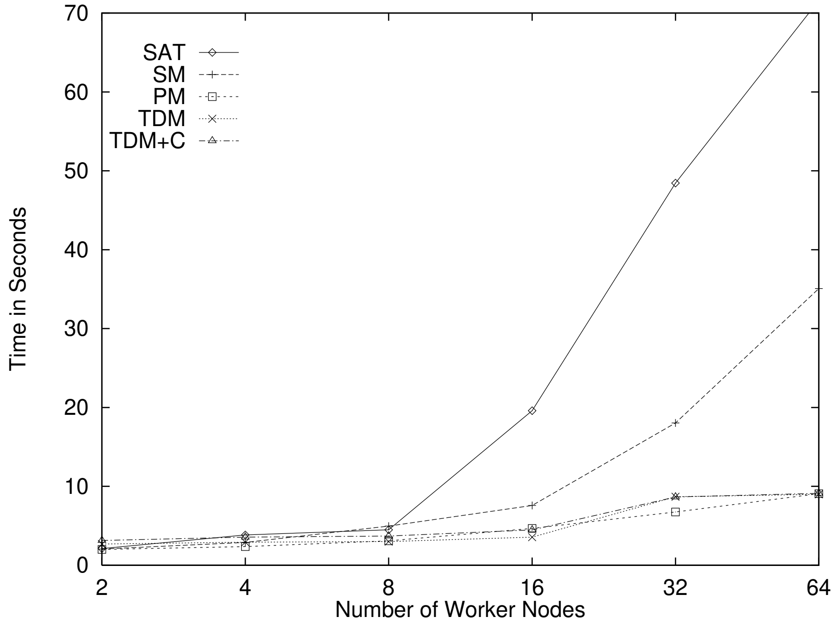

Figure 14: Scale-Up Results (4M tuple Dataset with 100% Reduction and SSN Partitioning)

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 impacted by the amount of data reduction as much as either PM or TDM+C which end up performing as well as TDM.

4.5 Scale-Up Performance : No Reduction and Time Partitioning

This experiment is designed to measure the effect of time partitioning on the scale-up properties of the proposed algorithms. The settings for this experiment are summarized in Table 8.

Parameter	Actual Value
$NumProcessors$	2, 4, 8, 16, 32, 64
$Partitioning$	by Time
$TupleSize$	41 bytes
$PartitionSize$	65536 tuples
$NumTuples$	(2,4,8,16,32,64)*65536
$DataReduction$	0 percent

Table 8: Parameters for the Time Partitioning with No Reduction Scale-Up Experiment

The dataset for this experiment was generated in a manner similar to the first one, but with StartDate rather than SSN partitioning. This was done by sorting the whole dataset by the StartDate attribute and then splitting it into 64 partitions of 64K tuples each.

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

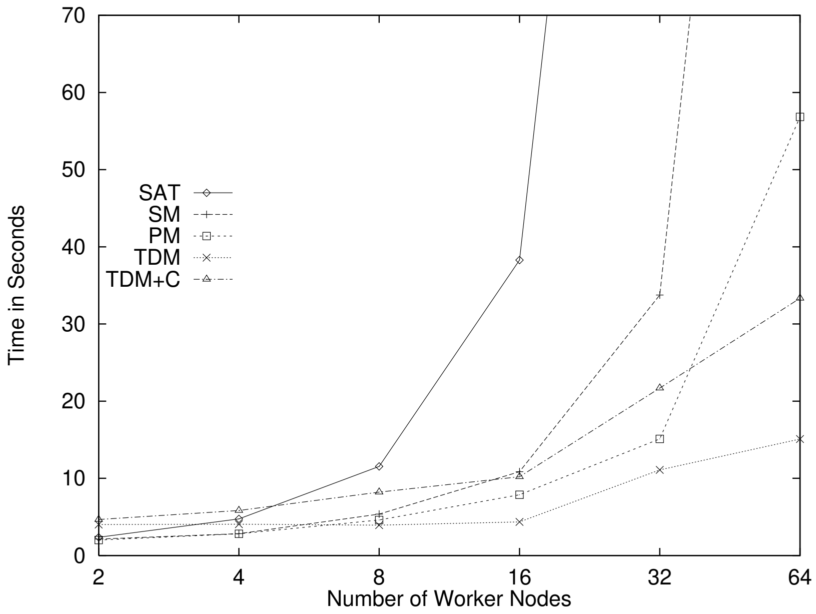

Figure 15: Scale-Up Results (4M tuple Dataset with No Reduction and StartDate Partitioning)

in all but the 16-node run. 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. 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.

4.6 Performance Measurement : Variable Reduction

This experiment is designed to measure the effect of a varying amount of data reduction on the scale-up properties of the proposed algorithms. The settings for this experiment, provided in Table 9, summarizes the parameters for this experiment.

Parameter	**Actual Value**
$NumProcessors$	16
$Partitioning$	by StartDate
$TupleSize$	41 bytes
$PartitionSize$	65536 tuples
$NumTuples$	(16)*65536
$DataReduction$	0/20/40/60/80/100 percent

Table 9: Parameters for the Variable Reduction with StartDate Partitioning Experiment

For this experiment, six sets of partitions were generated. Each set had 16 partitions, one for each of the 16 processing nodes participating in the six runs. The partitions were generated having 0, 20, 40, 60, 80 and 100 percent reduction. Timing results for this experiment are plotted on Figure 16 and leads us to the following observations.

TDM is the least affected by varying data reduction. The low slope of TDM's performance curve in Figure 16 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

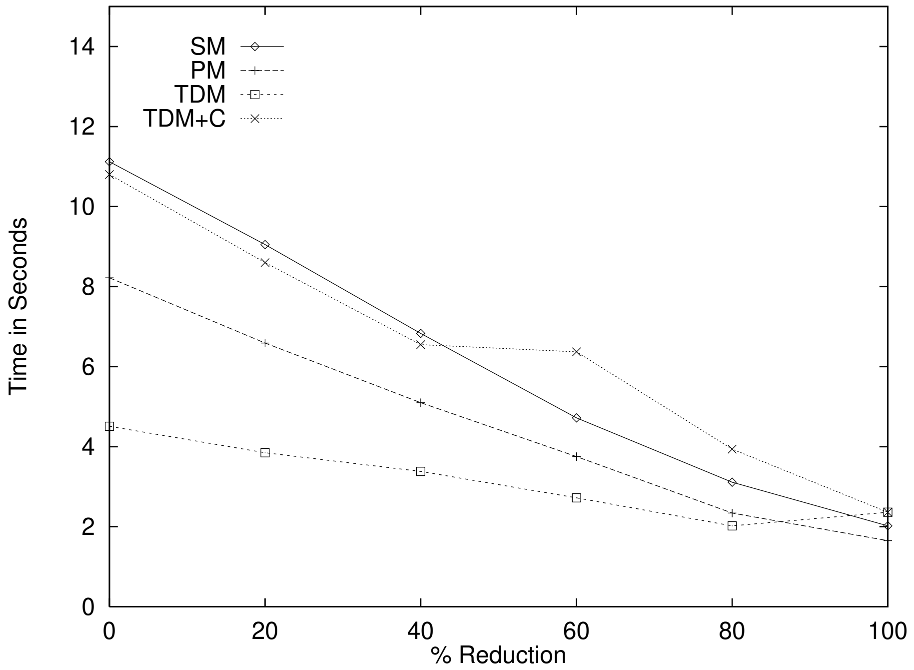

Figure 16: Variable Reduction Experiment (65536 tuples/node, 16 nodes, StartDate Partitioned)

we observed that the performance for TDM+C and PM caught up with TDM in the second experiment.

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 and fewer leaves were exchanged between nodes.

4.7 Summary

The empirical observations confirm that dataset partitioning, result placement, data reduction effected by the aggregation, and the number of processing nodes affect the proposed algorithms in different ways. SAT and SM, as seen in Figures 13, 14, and 15, were affected most by the number of processing nodes. Figure 16 shows that SM, SAT, PM and TDM+C were significantly affected by low data reduction while TDM was the least affected. Also, Figures 13, 14, and 15 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 scales well only when data reduction is high, as seen in Figure 14. TDM+C also provides centralized result placement but does not scale-up better than PM unless there is low reduction and the number of processing nodes is large. Lastly, dataset partitioning only affected the TDM variants, and even then, not substantially.

5 Conclusions

Temporal aggregate computations are important operations in a temporal database system. Traditionally, this has been an expensive operation in sequential database systems, 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 performance measurements and observed how different properties affected their behavior. 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 10.

Result Placement Methodology	Node Config. Size	Hi Reduction Dataset	Low Reduction Dataset
Distributed	Small	TDM	TDM
	Large	TDM	TDM
Centralized	Small	PM	PM
	Large	PM	TDM+C

Table 10: Matrix of Recommendations

1. Use TDM whenever distributed result placement suffices, regardless of any other parameter. As discussed in Section 3, distributed result placement is useful for distributed subqueries which are parts of larger distributed queries. Also, distributed result placement suffices when the aggregation results are not required for the *entire* timeline (i.e., finding the (time-varying) salaries of all employees for the last year).

2. For centralized result placement, use PM whenever there is a high degree of data reduction, regardless of the number of processing nodes. Also, use PM for centralized results, if there is low data reduction and the processing node configuration has 32 nodes or less.

3. For centralized result placement, low data reduction, and larger processing node configurations, use TDM+C.

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

1. *Improved algorithm for assigning time divisions to partitions.* We originally expected, StartDate partitioning to help TDM and TDM+C significantly. However, we discovered that the random partition assignment strategy that we employed did not exactly match TDM's naive time division assignment strategy, which ended up redistributing most of the leaves among the nodes. We still feel that StartDate partitioning will help the TDM variants perform better but we need a better time division assignment policy that attempts to minimize the number of leaves redistributed among the processing nodes.

2. *Impact of skew.* We expect PM to outperform TDM+C in queries with heavy tuple placement skew and/or selection skew [13]. Tuple placement skew occurs when the number of tuples are not evenly distributed physically amongst all the worker nodes before a query is initiated. Selection skew happens when some nodes return more candidate tuples than other nodes. 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. However, the specific impact of skew should be investigated.

3. *Aggregate functions.* Our experiments only considered scalar aggregates, which provide a single result at each point in time. Aggregate functions, for example listing the average salary by rank, should also be parallelized. We hypothesize that the algorithms that excel in small configurations would also work well with aggregate functions.

4. *Load balancing.* As mentioned in the empirical section, 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 for balancing the loads among the nodes would help reduce idle-waiting and improve parallel algorithm performance.

5. *Real-world Dataset.* All the experiments we have conducted so far have been on synthetic datasets. We therefore feel that testing the parallel algorithms on an actual dataset would provide a better understanding of how the parallel algorithms will perform in a realistic setting.

6. *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 will allow the algorithms to handle larger dataset sizes.

7. *Other Parallel Architectures.* All of our experiments were done on a shared-nothing Pentium cluster. The algorithms should also be studied on other multi-node architectures, such as shared disk and shared memory configurations, to ascertain the performance tradeoffs of these algorithms under alternate conditions.

References

[1] D. Bitton, H. Boral, D. DeWitt, and W. K. 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] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1), March 1997.

[4] Anindya Datta, Bongki Moon, and Helen Thomas. A case for parallelism in data warehousing and OLAP. In *International Workshop on Data Warehouse Design and OLAP Technology (DWDOT98)*, Vienna, Austria, August 1998.

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

[6] 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.

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

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

[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] R. T. Snodgrass, S. Gomez, and E. McKenzie. Aggregates in the temporal query language TQuel. *IEEE Transactions on Knowledge and Data Engineering*, 5:826–842, October 1993.

[11] 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.

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

[13] 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.