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

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 outpaced 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. For distributed results placement, we discovered that Time Division Merge was the obvious choice. For centralized results and high data reduction, Pairwise Merge was preferred regardless of the number of processing nodes, but for low data reduction, it only performed well up to 32 nodes. This led us to a centralized variant of Time Division Merge which was best for larger configurations having low data reduction.

1. Introduction

Aggregate functions are an essential component of data query languages, and are heavily used in many applications such as data warehousing. 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. 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.

In this paper, we present several new parallel algorithms for the computation of temporal aggregates on a shared-nothing architecture [8]. Specifically, we focus on the Aggregation Tree algorithm [7] and propose several approaches to parallelize it. The performance of the parallel algorithms relative to various data set and operational characteristics is of 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 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 gives an outlook to future work.

2. Background and Related Work

Simple algorithms for evaluating scalar aggregates and aggregate functions were discussed by Epstein [5]. A different approach employing program transformation methods to systematically generate efficient iterative programs for aggregate queries has also been suggested [6]. Tumas extended Epstein's algorithms to handle temporal aggregates [9]; these were further extended by Kline [7]. 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 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

[*]This work was sponsored in part by National Science Foundation grants CDA-9500991 and IRI-9632569, and National Science Foundation Research Infrastructure program EIA-9500991. The authors assume all responsibility for the contents of the paper.

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

Table 1. Sample Database and Its Temporal Aggregation

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 [4] implemented similar scalar aggregates and aggregate functions on a shared-nothing architecture. More recently, parallel algorithms for handling temporal aggregates were presented [11], 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 [7]. 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 [7], 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 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).

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.

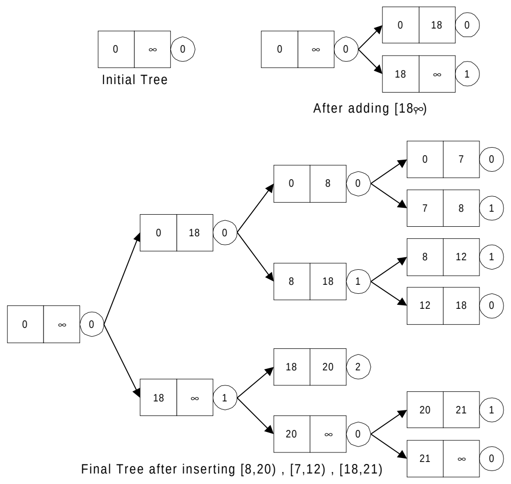


Figure 1. Example run of the Sequential(SEQ) Aggregation Tree Algorithm

3. Parallel Processing of Temporal Aggregates

In this section, we propose five parallel algorithms for the computation of temporal aggregates. We start with two simple parallel extensions to the SEQ algorithm, the Single Aggregation Tree (abbreviated SAT) and Single Merge (SM) algorithms. We then go on to introduce the Time Division Merge with Centralizing step (TDM+C) and Pairwise Merge (PM) algorithms, which both require more coordination, but are expected to scale better. 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.

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, and sends these periods up 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.

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, and 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. 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 2). The main steps for this algorithm are outlined in Figure 3.

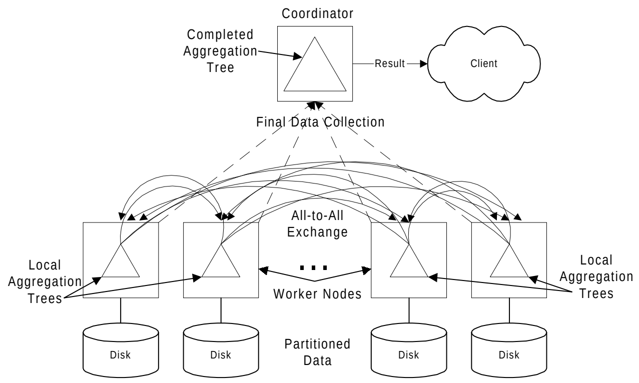


Figure 2. Time Division Merge with Centralizing Step (TDM+C) Algorithm

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
Step 6. Merge local results
Step 7. Return results to client

Figure 3. Major Steps for the TDM+C Algorithm

3.3.1 Overall Algorithm

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

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 periods 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.

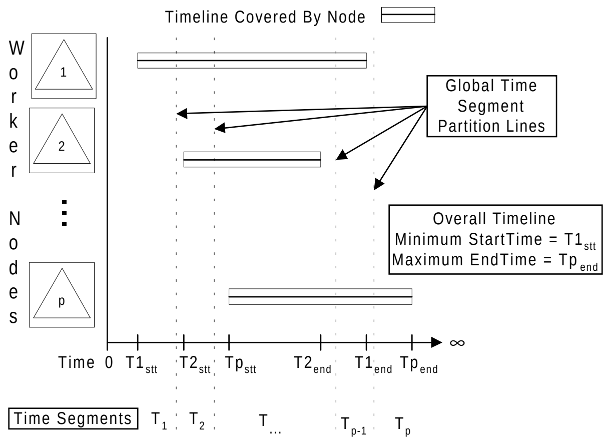

Timeline Covered By Node
W o r k e r N o d e s
1
2
p
Global Time Segment Partition Lines
Overall Timeline
Minimum StartTime = T1stt
Maximum EndTime = Tpend
Time 0 T1stt T2stt Tpstt T2end T1end Tpend ∞
Time Segments T1 T2 T... Tp-1 Tp

Figure 4. Timeline divided into p partitions, forming a global partition set

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

The coordinator takes all p local partition sets[1] and computes p global partitions (how this is done is discussed 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 periods which span more than one global partition period are split and each part is assigned accordingly(split periods do not affect the result correctness).

Each worker node merges the leaves it receives with the leaves it already has to compute the temporal aggregate for their assigned global partitions. 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.

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

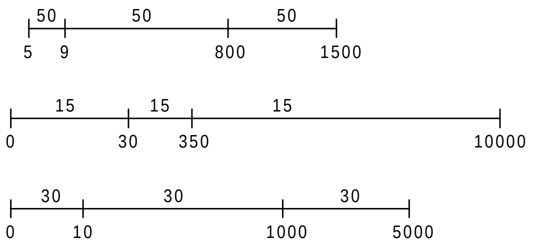


Figure 5. Local Partition Sets from Three Worker Nodes

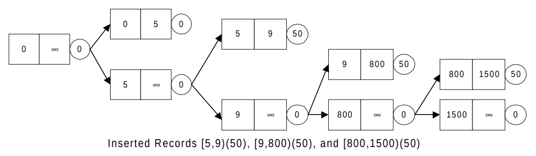
3.3.2 Calculating the Global Partition Set

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. The goal is to temporally distribute the computation of the final result, with each node processing roughly the same number of leaf nodes.

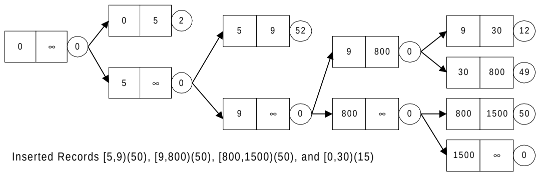
As an example, Figure 5 presents 9 local partitions from 3 worker nodes. The number between each hash mark segmenting a local timeline represents the number of leaf nodes within that local partition. The total number of leaf nodes from the 3 nodes is $50 \cdot 3 + 15 \cdot 3 + 30 \cdot 3 = 285$. The best plan is having $\frac{285}{3} = 95$ leaf nodes to be processed by each node. Figure 4 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 5, 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.

There are three main differences between our Modified Aggregation Tree algorithm used in this portion of TDM+C and the original Aggregation Tree [7], used in step 2 of Figure 3. First, the "count" field of this aggregation tree node is incremented by the count value of the local partition being inserted, rather than 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.



(a) First 3 Local Partitions


Inserted Records [5,9)(50), [9,800)(50), [800,1500)(50), and [0,30)(15)

(b) After partition 4 is added

Figure 6. Intermediate Aggregation Tree

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 6a. 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 $\frac{(5-0)}{(30-0)} \cdot 15 = 2$, while the right sub-tree's share is $15 - 2 = 13$. In this case, the left sub-tree leaf node [0,5) now has a count of 2 (see Figure 6b). We now pass 13 down the root right sub-tree, increasing its right leaf node count from [5,9)(50) to [5,9)(52) as its share of the newly added partition's count, 2, is added, by using the same proportion calculation method. At leaf node [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 $\frac{(30-9)}{(800-9)} \cdot 50 = 1$ 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 6b for the result. Table 2 shows the leaf node values once all 9 local time partitions from Figure 5 are inserted.

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 2. All leaf node values in a tabular format once all 9 partitions from Figure 5 are inserted

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

As an example, refer to Table 2. 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 [0,28), [28,866), and [866,10000).

The reader should be aware that this global time partition resolution algorithm is not perfect. The actual number of local aggregation tree leaves assigned to each worker node may not be identical. The reason is that the algorithm uses the local partition sets, which are just guides for the global partitioning. When a local partition has 50 leaf nodes in period [9,800), the global partition scheme assumes a uniform distribution, while the actual leaf nodes distribution may be heavily skewed.

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 two global synchronization steps that may limit the performance obtained. First, all of the local partition sets must be completed before the global time set partitioning can begin. Second, all of the worker nodes must complete their merges and send their results to the coordinator before the client can receive the final result.

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

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 7. Major Steps for the PM Algorithm

3.4. Pairwise Merge (PM)

The fourth parallel algorithm, PM (see Figure 7), 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.

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 8 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, $O(n)$, in the number of leaf nodes to be merged.

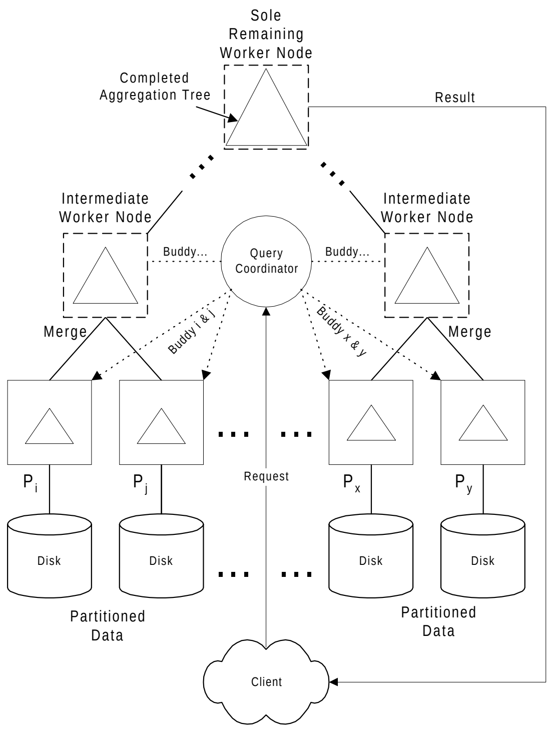


Figure 8. Pairwise Merge (PM) Algorithm

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 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` since it does not require that the attribute itself be sent. This simplifies 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. The matrix in Table 3 summarizes the experiments we have done.

	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 3. Experimental Case Matrix Summary

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 a 2GB hard disk. The machines were physically mounted on two racks of 32 machines. 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

To help precisely define the parameters for each set of tests, we established an experiment classification scheme. Table 4 lists the different parameters, and the set of parameter values for each experiment.

Synthetic datasets were generated to model relations which store time-varying information for each employee in a database. Each tuple has three attributes, an 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 data generation method 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 [3].

The *tuple size* was fixed at 41 bytes/tuple. The tuple size was intentionally kept small and unpadded so that the gener-

Parameter	Exp4.3	Exp4.4	Exp4.5	Exp4.6
$NumProcessors$	2, 4, 8, 16, 32, 64	2, 4, 8, 16, 32, 64	2, 4, 8, 16, 32, 64	16
$Partitioning$	by SSN	by SSN	by StartDate	by StartDate
$TupleSize$	41 bytes	41 bytes	41 bytes	41 bytes
$PartitionSize$	65536 tuples	65536 tuples	65536 tuples	65536 tuples
$NumTuples$	$NumProcessors$*65536	$NumProcessors$*65536	$NumProcessors$*65536	16*65536
$DataReduction$	0 %	100 %	0 %	0/20/40/60/80/100 %

Table 4. Experiment Parameters

ated 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.

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 with subsequent experiments. The second column of Table 4 gives the parameters for this particular 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] and 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 9 and lead us to the following conclusions.

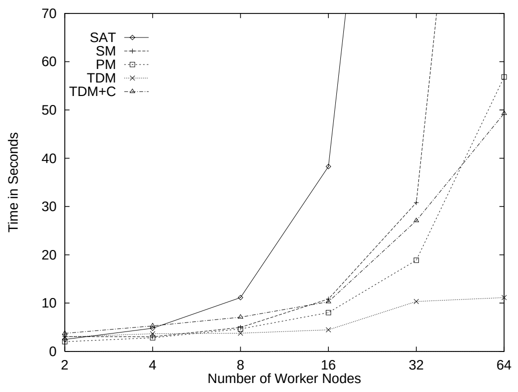


Figure 9. Scale-Up Results (4M tuple Dataset with No Reduction and SSN Partitioning)

SM performs better than SAT. Intuitively, since the dataset exhibits no reduction, both SM and SAT send *all* periods from the worker nodes to the coordinator. The reason behind SM's performance advantage comes from the computational parallelism provided by building local aggregation trees on each worker node. Aside from potentially reducing the number of leaves passed on to the coordinator, this process of building local trees sorts the periods in temporal order. 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. The performance difference increases with the number of nodes

[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.

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

[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.

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 4 gives the parameters for this 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 10 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.

[6]The aggregation tree algorithm performs at its worst case when the dataset is sorted by time [7].

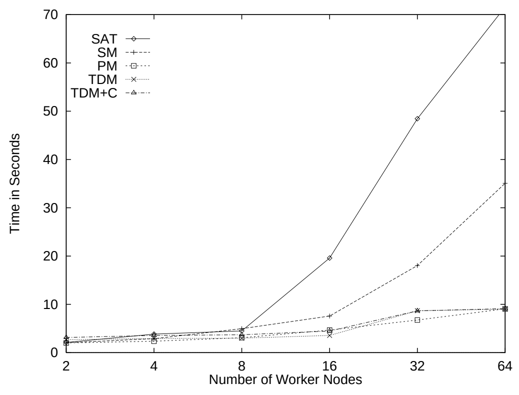


Figure 10. Scale-Up Results (4M tuple Dataset with 100% Reduction and SSN Partitioning)

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 4.

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 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.

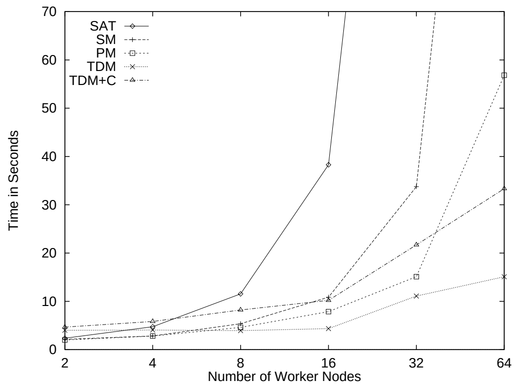


Figure 11. Scale-Up Results (4M tuple Dataset with No Reduction and StartDate Partitioning)

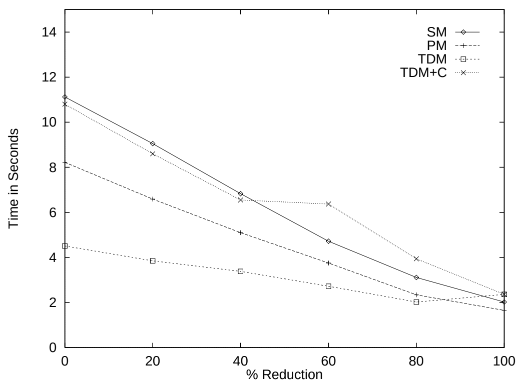


Figure 12. Variable Reduction Experiment (65536 tuples/node, 16 nodes, StartDate Partitioned)

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 4, summarizes the parameters for this 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 12 and lead us to the following observations.

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

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.

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 9, 10, and 11, were affected most by the number of processing nodes. Figure 12 shows that SM, SAT, PM and TDM+C were significantly affected by low data reduction while TDM was the least affected. Also, Figures 9, 10, and 11 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 10. 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 5.

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

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

Table 5. Matrix of Recommendations

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. Also, for a small configuration of processing nodes, having relatively high reduction, PM should be used.

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 global partitions to nodes.* For the TDM variants, we currently assign global partitions to worker nodes in a naive manner. We therefore need a better global partition assignment policy that attempts to minimize the number of leaves redistributed. In effect, this would lower network costs and improve performance.

2. *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 [10]. However, the specific impact of skew should be investigated.

3. *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.

4. *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.

5. *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.

6. *Deeper sensitivity analysis to other factors.* We have studied the effects of different parameters on the proposed algorithms. It is obvious, however, that other factors such as long-lived tuples and data distribution, among others, would affect the proposed techniques. Studying these effects is a ripe area for further research.

6 Acknowledgement

We would like to thank Minseok Park for his great work on generating performance numbers for this research effort.

References

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

[2] O. S. Center. LAM/MPI parallel computing. http://www.-osc.edu/lam.html, 1998.

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

[4] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar. 1990.

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

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

[7] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *the 11th Inter. Conference on Data Engineering*, pages 222–231, Taipei, Taiwan, Mar. 1995.

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

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

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

[11] X. Ye and J. A. Keane. Processing temporal aggregates in parallel. In *IEEE Inter. Conf. on Systems, Man, and Cybernetics*, pages 1373–1378, Orlando, FL, Oct. 1997.