Scalable Algorithms for Large-Scale Temporal Aggregation *

Bongki Moon[†] *Ines Fernando Vega Lopez*[†] *Vijaykumar Immanuel*[‡]

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

[‡]*Tandem Computers Inc.*
19333 Vallco Pkwy
Cupertino, CA 95014
vijay.immanuel@tandem.com

Technical Report 98-11

Abstract

The ability to model time-varying natures is essential to many database applications such as data warehousing and mining. However, the temporal aspects provide many unique characteristics and challenges for query processing and optimization. Among the challenges is computing temporal aggregates, which is complicated by having to compute temporal grouping. In this paper, we introduce a variety of temporal aggregation algorithms that overcome major drawbacks of previous work. First, for small-scale aggregations, both the worst-case and average-case processing time have been improved significantly. Second, for large-scale aggregations, the proposed algorithms can deal with a database that is substantially larger than the size of available memory. Third, the parallel algorithm designed on a shared-nothing architecture achieves scalable performance by delivering nearly linear scale-up and speed-up. The contributions made in this paper are particularly important because the rate of increase in database size and response time requirements has out-paced advancements in processor and mass storage technology.

November 1998

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

*This work was sponsored in part by National Science Foundation Research Infrastructure program EIA-9500991. The authors assume all responsibility for the contents of the paper.

Name	Salary	Dept	Begin	End
Richard	46,000	Accounting	18	31
Karen	45,000	Shipping	8	20
Nathan	35,000	Marketing	7	12
Nathan	38,000	Accounting	18	21

(a) Input Database Tuples

Count	Max	Begin	End
1	35,000	7	8
2	45,000	8	12
1	45,000	12	18
3	46,000	18	20
2	46,000	20	21
1	46,000	21	31

(b) Temporal Aggregation Results

Figure 1: Sample Database and Its Temporal Aggregation

1 Introduction

Aggregate functions compute a scalar value, such as the maximum salary, when applied to a set of tuples. These functions are an essential component of database query languages, and are heavily used in many applications. Several prominent query benchmarks such as TPC-D [18] and AS^{3}AP [20] contain a high percentage of aggregate operations. Hence, efficient execution of aggregate functions is an important goal.

Database applications often need to capture the time-varying nature of an enterprise they model. The importance of such need has been recognized by several database research groups, and temporal database models and query languages have been developed and reported in the literature [11, 17]. In fact, there are several temporal query languages supporting temporal aggregation [14, 15]. However, temporal data and queries provide many unique characteristics and challenges for query processing and optimization. Among the challenges is computing temporal aggregates, which is complicated by having to compute *temporal grouping*.

In temporal databases, temporal grouping is a process where the time-line is partitioned over time and tuples are grouped over these partitions. Then, aggregate values are computed over these groups. In general, temporal grouping is done by two types of partitioning [14]: *span grouping* and *instant grouping*. Span grouping is based on a defined length in time, such as week or month, and is independent of temporal attribute values of database tuples. On the other hand, instant grouping depends on the data stored. Any pair of consecutive instants create a time interval, over which the aggregate value remains constant. Such intervals are called constant intervals. Aggregations based on span and instant groupings are called *span aggregation* and *instant aggregation*, respectively. In this paper, we focus on computing instant aggregates, which we believe is the most common and challenging temporal aggregation.

Computing instant aggregates is expensive because it is necessary to know which tuples overlap each instant, and simply considering each tuple in order in a sorted-by-time relation will not be sufficient due to the varying interval lengths [12]. For example, computing the time-varying maximum salary of employees involves computing the temporal extent of each maximum value, which requires determining the tuples that overlap each temporal instant. Figure 1(a) shows a sample `Employees` table with two temporal attributes, which represent the beginning and ending of the valid-times of individual tuples. The resulting instant aggregation of the maximum salary (along with the number of employees) is given in the table in Figure 1(b). Note that while multiple values are returned, the aggregate results in a single scalar value at each point in time, with the period over which the aggregate value remains constant collected into a single tuple. One could also envision an instant aggregate function, which would evaluate a time-varying maximum salary for each department.

This temporal aggregation can be processed in a sequential or parallel fashion. The parallel processing technology becomes even more attractive, as the size of data-intensive applications grows as evidenced in OLAP and data warehousing environments [4, 6]. Although several sequential and parallel algorithms have been developed for computing temporal aggregates [10, 12, 15, 19, 21], they suffer from serious limitations such as the size of aggregation restricted by available memory and requirement of a priori knowledge about the orderedness of an input database.

In this paper, we propose a variety of temporal aggregation algorithms that overcome major drawbacks of previous work. The proposed solutions provide the following benefits over the state of the art:

- Two new algorithms proposed for small-scale aggregations do not require a priori knowledge about an input database, and they have improved both the worst-case and average-case processing time significantly.

- Another new algorithm proposed for large-scale aggregations relies on a novel data partitioning scheme, so that it can deal with a database substantially larger than the size of available memory.

- Lastly, a parallel algorithm has been developed for shared-nothing architectures for large-scale aggregations. This solution achieves scalable performance by delivering nearly linear scale-up and speed-up.

It should be noted that the problem of computing temporal aggregates is different from the relational aggregation that can often be seen in the data warehousing environment. While data items in the data warehousing environment are envisioned as points in their data domain, we deal with temporal data associated with time intervals of arbitrary lengths.

The rest of this paper is organized as follows. Section 2 surveys the background and related work on computing temporal aggregates. Major Limitations of previous work are also discussed in the section. In Sections 3, 4 and 5, we present the improved algorithms for small-scale aggregations, and scalable solutions for large-scale aggregations based on data partitioning and parallel processing techniques. Section 6 presents the results of experimental evaluation of the proposed sequential and parallel solutions. Finally, Section 7 summarizes the contributions of this paper and gives an outlook to future work.

2 Background and Previous Work

There are two types of aggregate computations in conventional relational database systems: scalar aggregates and aggregate functions. Scalar aggregates are operations such as `count`, `sum`, `avg`, `max`, and `min` 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.

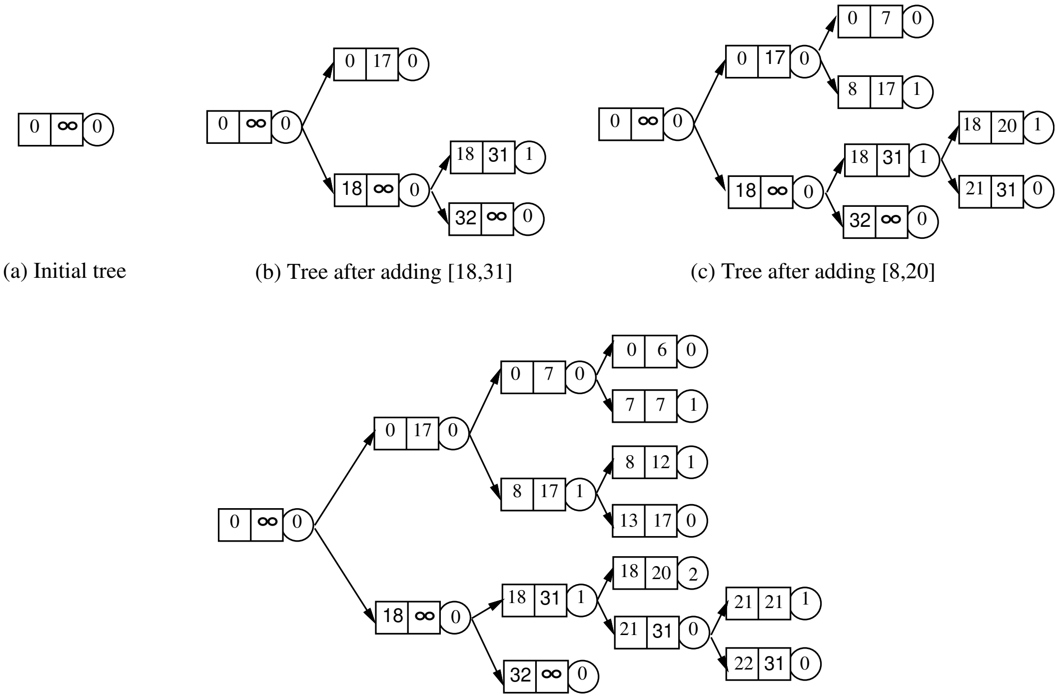
A scalar aggregate is composed of an aggregate expression and an optional qualification. A simple two-step algorithm was proposed by Epstein for evaluating scalar aggregates [8]. To handle many scalar aggregates in a query, the algorithm computes each of them separately and stores each result in a singleton relation, referring to that singleton relation when evaluating the rest of the query. A different approach employing program transformation methods was proposed to systematically generate efficient iterative programs for aggregate queries [9].

The first approach for implementing temporal aggregation was proposed by Tuma [19] and was based on an extension of Epstein's algorithm. In this approach, the constant intervals are determined first, then the aggregate is evaluated using the Epstein's technique. Since the two steps are separate and the first one must be completed before the second one, a database must be read twice.

More recent algorithms were proposed by Kline and Snodgrass [12] for temporal aggregation based on instant grouping of tuples. The algorithms are called *aggregation tree* and its variant *k-ordered aggregation tree*, as they build a tree while scanning a database. Both algorithms are fast and require minimal I/O overhead, as they need to scan the database only once to build a tree in memory. Then, the resulting tree stores enough information to compute temporal aggregates by traversing it using depth first search.

The aggregation tree is a binary tree that tracks tuples whose timestamp periods contain an indicated time span. Each node of the tree contains a start time, an end time and an aggregate value. When an aggregation tree is initialized, it begins with a single node containing $< 0, \infty, 0 >$ (see the initial tree in Figure 2), assuming that 0 and ∞ are used as the earliest and latest timestamps. The sample `Employees` table in Figure 1(a) has 4 tuples to be inserted into the empty aggregation tree. Inserting the first record adds four new nodes to the initial tree, resulting in the updated aggregation tree shown in Figure 2(b). A count of one is assigned to the new leaf $< 18, 31, 1 >$, since it is the only node in the tree representing a valid interval for the inserted tuple. The aggregation tree after inserting the rest of the records in Figure 1(a) is shown in Figure 2(d).

To compute the number of tuples (*i.e.*, `count` aggregate) for the period $[8, 12]$ in this example, The count from the leaf node $[8, 12]$ (which is 1) is added to 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).



(a) Initial tree (b) Tree after adding [18,31] (c) Tree after adding [8,20]

(d) Final tree after adding [7,12] and [18,21]

Figure 2: A Sample Aggregation Tree for `count` Aggregation

2.1 Limitations of Previous Methods

It should be noted that the order of tuples inserted into the aggregation tree affects its performance, though 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. For the reason, the worst case time to create an aggregation tree is $\mathcal{O}(N^2)$ for N tuples sorted in time. However, more serious limitation of the aggregation tree approach is that the entire tree must be kept in memory. Since the size of an aggregation tree is proportional to the number of distinct timestamps (both start times and end times), the size of the database the aggregation tree algorithm can deal with tends to be limited by the size of available memory and the number of distinct timestamps of tuples.

To circumvent this problem, a variant of the aggregation tree, called k-ordered aggregation tree, was proposed by the same authors. The k-ordered aggregation tree takes advantage of the *k-orderedness* of tuples to enable garbage collection of tree nodes, so that the memory requirements can be reduced significantly. However, the k-ordered aggregation tree approach assumes that the tuples in a table be ordered within a certain degree. Specifically, each tuple is at most k positions from its position in a totally ordered version of the table. This requirement is difficult to be met in a real database system. Without a priori knowledge about a given table, the k-orderedness is expensive to measure, as it requires an external sort of the table. The worst case running time of the k-ordered aggregation tree algorithm is still $\mathcal{O}(N^2)$.

Apparently the aggregation tree, the most efficient among the aforementioned algorithms, suffers from poor scale-up performance, due to the $\mathcal{O}(N^2)$ worst-case running time and memory requirement. Recently, there have been some research efforts to develop parallel algorithms for computing temporal aggregates for large-scale databases. Ye and Keane proposed two approaches to parallelize the aggregation tree algorithm on a shared-memory architecture [21]. Gendrano *et al.* have also developed several new parallel algorithms [10] for computing temporal aggregates, specifically on a shared-nothing architecture, by parallelizing the aggregation tree algorithm. Gendrano *et al.* showed promising scale-up performance of the parallel algorithms through extensive empirical studies under various conditions. Nonetheless, all the aforementioned parallel algorithms inherit the same limitations from the aggregation tree algorithm, as the parallel algorithms were developed by parallelizing the aggregation tree. In particular, the size of the database those parallel algorithms can handle will be limited by the aggregate memory of participating processors.

3 Improved Algorithms for Small-Scale Aggregation

In this section, we present two new algorithms for computing temporal aggregates, as alternatives to the aggregation tree algorithm [12]. The aggregation tree is a binary tree, which is similar to the segment tree by Bentley [2]. The segment tree is a static structure, which can be balanced for a given set of abscissae. However, there is no guarantee that the aggregation tree is always balanced, because the aggregation tree is dynamically constructed as the tuples in a database are being scanned and inserted into the tree. Thus, the structure of the resulting aggregation tree depends on the order of tuples inserted. This fact may cause the worst case running time of $\mathcal{O}(N^2)$ for a database of N tuples, particularly when the tuples are ordered by their timestamp values. Such a quadratic complexity may be impractically costly for many database applications.

As will be seen in this section, we have observed that the five most common aggregation operators can be categorized into two groups, namely, `count`, `sum`, `avg` in one group, and `max`, `min` in the other. For the latter group, there is more demand to keep track of attribute values of tuples. This observation has led us to develop a different algorithm for each of the two groups of aggregation operators. The solution to the first group of operators, which we call a *balanced tree* algorithm, will be presented in Section 3.1. The main idea of this algorithm is that the tree can be balanced dynamically as tuples are being inserted, by giving up the notion of maintaining intervals in the tree nodes. The solution to the second group is called a *merge-sort aggregation* algorithm, which is similar to the classical merge-sort algorithm [13]. This algorithm will be presented in Section 3.2. In this section, we assume that the memory is large enough to store the entire data structures required by each aggregation algorithm. In the rest of this paper, we use the `count` and `max` as the representatives of the two groups of operators, respectively.

3.1 Balanced Tree Algorithm for `count` Aggregation

A relatively simple approach based on timestamp sorting can provide an efficient solution for the `count` aggregation. This approach starts with loading the entire tuples in memory. Then, the timestamp values are extracted from the tuples, and each timestamp is associated with a tag, which indicates whether the timestamp is a start time or an end time of a tuple. These timestamps and tags are then sorted in an increasing order of the timestamp values. See Figure 3 for a sorted list of timestamps and tags for a sample database given in Figure 1(a).

Finally, the `count` aggregate is computed by scanning the sorted timestamps and tags in an increasing order. Getting started with a counter initialized to zero, the counter is incremented by one when a START tag is encountered, and it is decremented by one when an END tag is encountered. When more than one tags are associated with a timestamp, the counter is incremented by the number of START tags or decremented by the number of END tags. For example, in Figure 3, when the timestamp value 18 is encountered, the counter is incremented by two from 1 to 3 because there are two START tags associated with the timestamp. Apparently, the worst case processing time of this approach is $\mathcal{O}(N \log N)$, where N is the number of tuples in an input database.

In real world temporal databases, it may be the case that many tuples share the same timestamp values for their start times and end times. Nonetheless, this approach requires the same amount of memory and processing time regardless of the repeated timestamp values. Thus, we propose a *balanced tree* algorithm to further optimize its performance for such databases with repeated timestamp values.

The motivation behind the balanced tree algorithm is that the sorted list of timestamps can be built even without loading an entire database into memory at once. Instead, the timestamps can be sorted *incrementally* by inserting them into a balanced tree, as the tuples of an input database are being scanned. Each node of a balanced tree stores a timestamp, either a start time or an end time, but need not store a START/END tag. Instead of the tag, each node stores two counters: one storing the number of tuples starting at the

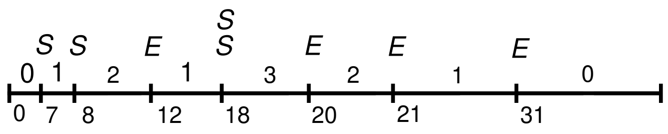


Figure 3: Example of `count` Aggregation by Sorting Timestamps and Tags

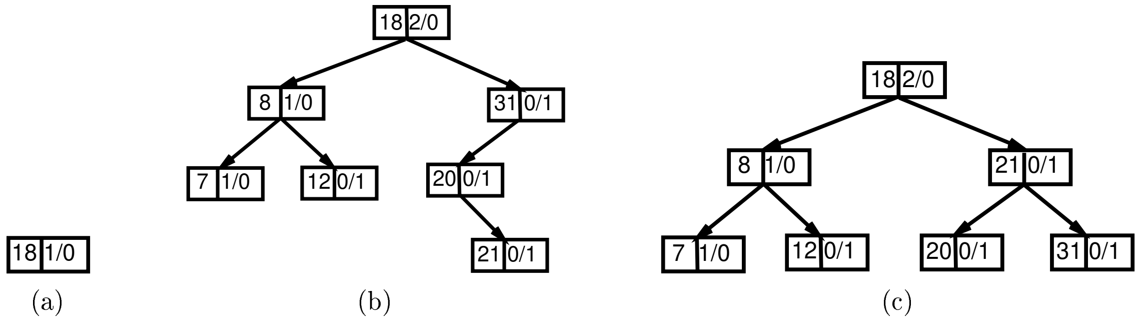


Figure 4: Example of Balanced Tree Construction

timestamp and the other storing the number of tuples ending at the timestamp. [1] Additionally, a color tag is stored in each node, as we use the *red-black* insertion algorithm [5] to keep the tree balanced dynamically.

Figure 4 shows the process of building a balanced tree for the sample `Employees` table in Figure 1(a). In the figure, we only show timestamps and counters, which are relevant to temporal aggregate computation. When the start time 18 of the first record is inserted into an empty tree, a new node is created for the timestamp, and then its start-counter and end-counter are set to one and zero, respectively. The resulting tree having a single node is shown in Figure 4(a). Figures 4(b) and (c) illustrate snapshots of the tree before and after the tree is balanced by the red-black insertion algorithm. We do not elaborate on the red-black insertion because it is not the focus of this paper.

The balanced tree algorithm proceeds in two steps, first by creating the tree and then by traversing the tree. Whenever a tuple is read from an input database, the balanced tree is probed to see whether the start and end times of the tuple are already in the tree. If the start (or end) timestamp is not found in the tree, then a new node is created and inserted into the tree. Otherwise, the start time (or end time) counter of a node that contains the timestamp is incremented by one without inserting a new node. Once the balanced tree has been built, the algorithm computes aggregate values while performing an in-order traversal of the tree. Specifically, whenever a tree node is visited, the `count` aggregate value is incremented by the start-counter value of the node and decremented by the end-counter value of the node. The proposed balanced tree algorithm is summarized in Figure 5.

By eliminating redundant timestamp values from the tree, the balanced tree algorithm reduces the memory requirements and tree traversal time substantially especially for a database with a small percentage of unique timestamps. The balanced tree stores information needed for temporal grouping and aggregation both in internal nodes and leaf nodes. Thus, the balanced tree algorithm uses only half the nodes required by the aggregation tree algorithm, which stores constant intervals only in leaf nodes.

[1] For `sum` aggregation, each node stores two variables: one storing the attribute value sum of the tuples starting at the timestamp and the other storing the attribute value sum of the tuples ending at the timestamp.

Algorithm 1 *Balanced Tree*

```
set 𝒯 ← an empty balanced tree;
for each tuple t in a table do begin
    if (t.start_time = n.ts for any node n in 𝒯) then n.no_starts++;
    else insert a new node n' (with n'.ts = t.start_time) into 𝒯;
    endif
    if (t.end_time = n.ts for any node n in 𝒯) then n.no_ends++;
    else insert a new node n' (with n'.ts = t.end_time) into 𝒯;
    endif
end
set count ← 0;
for each node n in 𝒯 traversed by in-order do begin
    count += n.no_starts;
    output n.ts and count;
    count -= n.no_ends;
end
end Algorithm
```

Figure 5: Balanced Tree Algorithm for `count` Aggregation

3.2 Merge-Sort Algorithm for `max` Aggregation

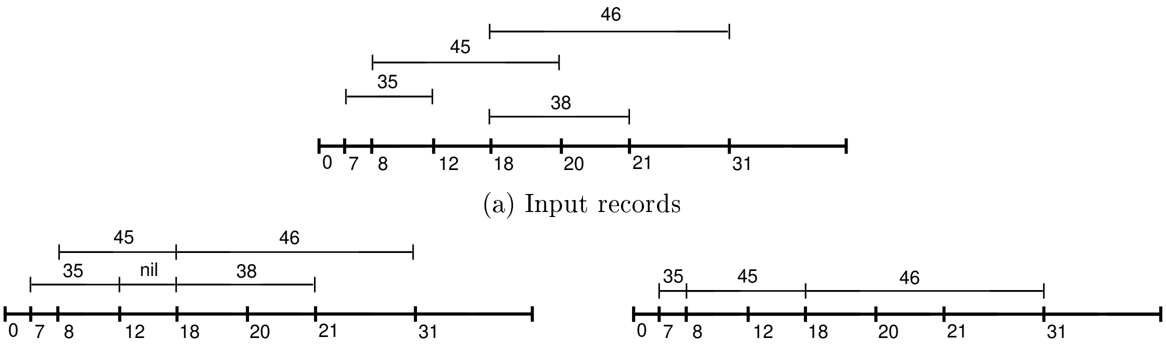
While the balanced tree algorithm is simple and efficient for `count` aggregations, it cannot be use for `max` aggregations. Since a balanced tree stores only unique timestamps and associated counters for `count` aggregation, it is not possible to keep track of all the tuples that are alive at a given time instant with the information available in the tree. For example, in Figure 4(c), the root node shows that there exist two tuples whose start times are 18. However, the tree does not convey any information about the life spans of the tuples (*i.e.*, the exact end times of the two specific tuples). Unlike `count` aggregations, it is impossible to compute `max` aggregations without knowing the exact life spans of tuples in a database.

One can modify the balanced tree algorithm to compute `max` aggregates, by allowing repeated timestamp values in a tree and using additional data structures such as dual heaps while traversing the tree. The dual heaps store the attribute values (on which the `max` aggregation is performed) of live tuples and dead tuples, separately. While traversing the tree, the `max` aggregate can be computed by comparing two maximum values in both the heaps and popping matched maximum values from the heaps. In fact, the dual heaps are used to keep track of the life spans of tuples that are required to compute the `max` aggregate. However, with this modification, we will lose all the benefits of using the balanced tree algorithm, because the tree will need exactly two nodes per each tuple (*i.e.*, no reduction in memory requirements due to repeated timestamps) and additional overhead for processing the heaps will be non-trivial.

Instead, we propose a *bottom-up* aggregation approach, which we call a *merge-sort aggregation* algorithm. Like the classical merge-sort algorithm based on the divide-and-conquer strategy, the merge-sort aggregation algorithm computes a larger (intermediate) aggregate result by merging two smaller (intermediate) aggregate results. The algorithm starts with merging tuples in pairs at the bottom and terminates when a final aggregate result is obtained at the top.

Formally, an intermediate aggregate can be defined as (T_k, M_k), where $T_k = \{t_0, t_1, \ldots, t_k\}$ and $M_k = \{m_1, m_2, \ldots, m_k\}$ for an integer $k \geq 1$. T_k is a set of $k+1$ unique timestamps in an increasing order $(t_0 < t_1 < \ldots < t_k)$. M_k is a set of k attribute values, where $m_i\,(1 \leq i \leq k)$ is a maximum attribute value associated with a time interval $[t_{i-1}, t_i)$ if there exist at least one live tuple in $[t_{1-1}, t_i)$. Otherwise, $m_i = nil$ for an empty interval. No two consecutive values in M_k are equal (*i.e.*, $m_i \neq m_{i+1}$ for any $i\,(1 \leq i \leq k-1)$). Each tuple t in an input database can be considered as a (T_1, M_1) with $T_1 = \{t.start\_time, t.end\_time\}$ and $M_1 = \{t.attribute\_value\}$.

Figure 6 illustrates the process of merging the tuples of the sample `Employees` table in Figure 1(a). The



(a) Input records

(b) After the first merging step

(c) After the second merging step

Figure 6: Example of Merging for `max` Aggregation

sample tuples are described as four line segments in Figure 6(a). In the first step, the first two tuples in the `Employees` table are merged into an intermediate result $(\{8, 18, 31\}, \{45000, 46000\})$; the last two tuples are merged into an intermediate result $(\{7, 12, 18, 21\}, \{35000, nil, 38000\})$. The result of the first step is shown in Figure 6(b). In the second step, the two intermediate results are merged together into the final aggregate result $(\{7, 8, 18, 31\}, \{35000, 45000, 46000\})$, as shown in Figure 6(c).

As an input database of N tuples is scanned, the merge-sort aggregation algorithm generates $\lceil N/2 \rceil$ first-step intermediate aggregates in memory. Then, the algorithm recursively merges the intermediate results until a final aggregate result is obtained. Thus, the worst case processing time of the algorithm is $\mathcal{O}(N \log N)$. As is shown in Figure 6, the size of an intermediate result (T_k, M_k) may be smaller than the tuples themselves covered by (T_k, M_k), because two consecutive intervals can be merged into a single interval if they share the same aggregate value (*i.e.*, maximum in the example). Thus, the amount of additional memory required for intermediate results is likely to be smaller than the size of an input database. Nonetheless, for `count` aggregations, the balanced tree will remain as the algorithm of choice. This is because the balanced tree algorithm will keep the memory requirement (*i.e.*, the number of tree nodes) down to the minimum by building a balanced tree incrementally and by removing repeated timestamps, and thereby minimizing its processing time.

4 Bucket Algorithm for Large-Scale Aggregation

In addition to the algorithms for small-scale aggregations proposed in the previous section, another major component of the work proposed in this paper is to develop new techniques for computing temporal aggregates under the constraint of limited buffer space. Then, the size of databases we can deal with is not limited by the size of available memory. Additionally, it is crucial that temporal aggregation require only a constant number (say, two or three) of database scans, due to potentially huge amount of temporal data. It will be prohibitively costly for a large-scale database, if the number of required database scans is not limited and is rather proportional to the size of database. For this reason, we do not consider as an acceptable solution any method that requires more than a small constant number of database scans.

In this section, we propose a new algorithm based on partitioning database tuples into several buckets, which has been used for many important database operations such as the relational hash join algorithm. The idea of the hash join algorithm is to hash two joining relations on the join attribute, using the same hash function. Then, it is assured that tuples of one relation in a bucket can join only with tuples of the other relation in the same bucket. Thus once both relations are partitioned, the join operation can be performed by reading the relations just once, provided that enough memory is available to keep all the tuples of one relation in a bucket in memory. Assuming uniform distribution of data, it has been shown that the hash

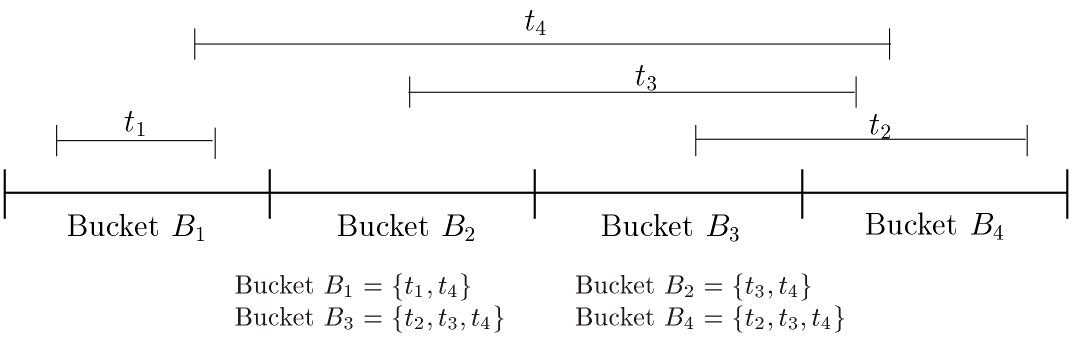


Figure 7: Time-line partitioning and assignment of tuples into buckets.

join algorithm requires three database scans if the number of buffer pages is larger than a square root of the number of disk pages in a smaller relation [7].

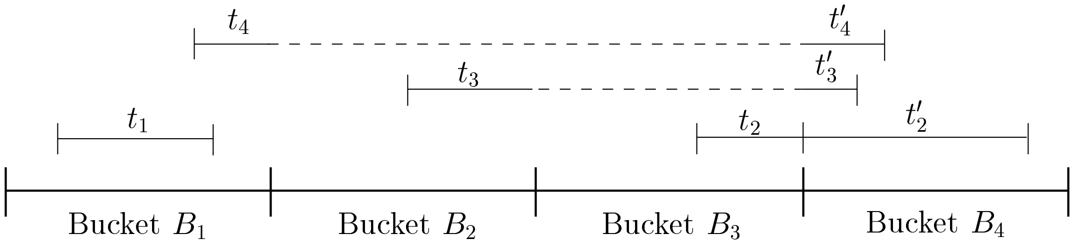
Although the idea of data partitioning appears promising for relational hash join operation, it cannot be applied directly to temporal aggregation. Tuples associated with time intervals are not readily partitioned into temporally disjoint equivalence classes (*e.g.*, hash buckets), because the time intervals of tuples may be of any length. Some tuples may overlap with the intervals of more than one buckets, and such tuples must be checked with tuples in all the overlapping buckets. That is, there is no guarantee that temporal aggregates can be computed by reading the buckets only a constant number of times.

To circumvent this problem, one can allow assignment of a data object into multiple buckets by replicating it. This approach can be best described by an example given Figure 7. The time-line of a given temporal database is partitioned into $\mathcal{N}_B$ disjoint intervals, where $\mathcal{N}_B$ is the number of buckets. If a tuple's life span is contained in the interval of a bucket, the tuple is assigned to the bucket. For example, in Figure 7, tuple t_1 will be assigned to bucket B_1 as t_1's life span is properly contained in that of bucket B_1. On the other hand, if a tuple's life span overlaps two or more intervals (say, k intervals), the tuple's life span is split into k pieces and these pieces may be assigned to k buckets. (It turns out that splitting a tuple into several does not impact the result of the aggregation.) In Figure 7, the life spans of tuples t_2, t_3 and t_4 overlap with 2, 3 and 4 buckets, respectively. Thus, tuple t_2 will be assigned to buckets B_3 and B_4, t_3 to buckets B_2, B_3 and B_4, and t_4 to buckets B_1, B_2, B_3 and B_4.

This process entails replicating tuples and may lead to considerable duplication of data, especially for long-lived tuples. To minimize duplication of tuples, we propose to assign each tuple solely to the buckets where the tuple's start and end timestamps lie. Suppose the life span of a tuple t overlaps buckets $B_i, B_{i+1}, \ldots, B_j (0 \leq i < j < \mathcal{N}_B)$. Then, the tuple t will be replicated only in the buckets B_i and B_j, but the intermediate buckets will not store the tuple t. Instead, a *meta array* is used to aggregate the information that the tuple t's life span overlaps the intermediate buckets $B_{i+1}, \ldots, B_{j-1}$. The size of a meta array is equal to the number of buckets. The i-th element of a meta array stores an aggregate value (*e.g.*, `count`) for the i-th bucket.

For example, in Figure 8, the time interval of tuple t_3 spans over three buckets B_2, B_3 and B_4. Thus, t_3 is split into two segments (*i.e.*, t_3 and t_3') with adjusted time intervals so that each segment can be properly contained in the interval of its corresponding bucket. (Solid lines in Figure 8 represent adjusted time intervals of split tuples.) Then, t_3 and t_3' are assigned to two buckets B_2 and B_4, respectively; the third element of the meta array is incremented by one. In a similar way, t_4 and t_4' are assigned to two buckets B_1 and B_4 respectively, and the second and third elements of the meta array are incremented by one. The resulting data partitioning and meta array are illustrated in Figure 8. Note that neither the first nor the last element of the meta array stores a valid aggregate value, as no tuple can have a life span longer than the time-line of an entire database.

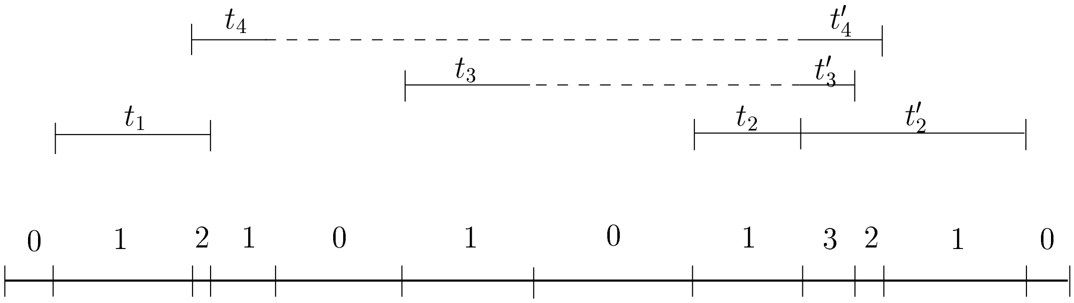
Once all the tuples are scanned and partitioned into buckets and a meta array is created, the temporal aggregate operation can be performed on each bucket independently. Figure 9(a) shows the partial results of the aggregation performed on each bucket. Then, each aggregate value stored in the meta array is com-



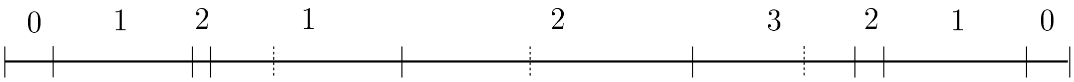
Bucket $B_1 = \{t_1, t_4\}$ Bucket $B_2 = \{t_3\}$
Bucket $B_3 = \{t_2\}$ Bucket $B_4 = \{t'_2, t'_3, t'_4\}$

Meta Array for `count` aggregation $= \{0, 1, 2, 0\}$

Figure 8: Meta Array and Reduced Data Replication



(a) After aggregated bucket by bucket



(b) After combined with the meta array

Figure 9: Steps of the aggregation based on data partitioning and meta array

bined with the aggregation results from each corresponding bucket (*e.g.*, simply by adding counts for `count` aggregation). Lastly, the final aggregation results can be obtained by merging each pair of adjacent buckets at their boundaries if the two adjacent aggregate values are equal. Figure 9(b) shows the final aggregation results. The dotted vertical bars in the figure represent the merged bucket boundaries. Figure 10 outlines the proposed temporal aggregation algorithm based on data partitioning.. In the algorithm description, it is assumed that the entire time-line of a table is partitioned into $\mathcal{N}_B$ disjoint intervals of an equal length, each of which is associated with a bucket. Note that any small-scale aggregation algorithm proposed in the previous section can be used to aggregate each individual bucket.

Provided that the meta array is small enough to fit in memory and sufficient memory is available to hold all the tuples in a bucket, the temporal aggregate operation can be performed by reading each bucket just once. Thus, in total, this approach requires three database accesses (*i.e.*, two reads and one write) to compute temporal aggregates. Considering the data replication for the tuples overlapped with multiple buckets, the database access requirement of this approach is likely to increase to some extent depending on

Algorithm 2 *Temporal Bucketization*

set $\mathcal{I}_B \leftarrow$ time interval for each bucket $((\mathcal{T}_{max} - \mathcal{T}_{min})/\mathcal{N}_B)$;
for each tuple `t` in a table **do begin**
  set `start_bucket` $\leftarrow$ (`t.start_time` $- \mathcal{T}_{min})/\mathcal{I}_B$;
  set `end_bucket` $\leftarrow$ (`t.end_time` $- \mathcal{T}_{min})/\mathcal{I}_B$;
  insert `t` into a bucket B_{start_bucket};
  if (`start_bucket` $\neq$ `end_bucket`) insert `t'` into a bucket B_{end_bucket};
  for (`i=start_bucket+1` **to** `end_bucket-1`) **do** update `meta_array[i]`;
end
for (`i=0` **to** $\mathcal{N}_B - 1$) **do begin**
  perform temporal aggregation on the bucket B_i;
  combine the scalar value of `meta_array[i]` to the bucket B_i;
  merge the bucket boundary with B_{i-1} as needed;
end
end Algorithm

Figure 10: Bucket Algorithm based on Temporal Data Partitioning

various factors such as the life spans of tuples and the number of buckets used. Even in the worst case, however, the size of a given table can increase only up to twice its original size by replicating each tuple in the table into two buckets. Thus, the database access requirement of this approach is still bounded to a small constant number of scans. We will show the performance impact of data replication in Section 6.

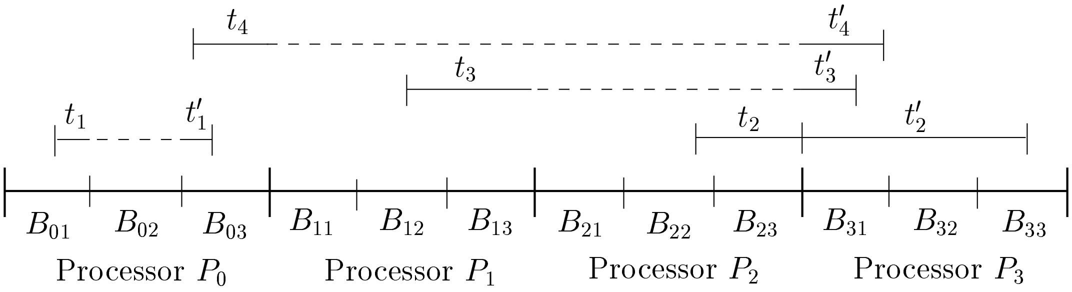
5 Parallel Bucket Algorithm

Parallel processing for database applications typically involves partitioning of data, followed by allocation of the partitions to a set of processors. Then, the processors perform operations on the partitioned data in parallel, achieving speed-up in query processing times. Among the various architectures that have been proposed for parallel database systems, a *shared-nothing* architecture [16] has made it an attractive choice for large-scale database applications due to its high potential for scalability. By scalability we mean the capability of delivering an increase in performance proportional to an increase in the number of participating processors.

In a shared-nothing architecture, each processor owns local memory and secondary storage units, and communicates each other by message passing. Initial data placement can be either centralized or distributed across multiple processors. For most of the parallel database operations, however, some of the data may have to be redistributed amongst processors that actually participate in the operations. We assume that resulting aggregates remain in local storage units of the participating processors without collecting the results on a special coordinator processor. Then, the resulting aggregates can be used as intermediate data for the next phase of parallel query processing.

As was pointed out in Section 2, most of the previous attempts to develop scalable methods for computing large-scale temporal aggregates were based on parallelizing the aggregation tree algorithm. For the reason, those approaches inherit all the limitations the aggregation tree algorithm has. Specifically, these approaches will suffer from $\mathcal{O}(N^2)$ worst-case running time and tight limitations on a database size they can deal with.

In this section, we propose a new parallel temporal aggregation algorithm based on the bucket algorithm (Algorithm 2) presented in the previous section. It is relatively straightforward to parallelize the bucket algorithm by distributing buckets across participating processors. The time-line of a given temporal database is partitioned into $\mathcal{P}$ disjoint intervals, where $\mathcal{P}$ is the number of processors. Then, on each processor, the time-line of the processor is again partitioned into $\mathcal{N}_B$ disjoint intervals. However, distributing the buckets is not enough to compute correct aggregate results, because the construction of meta arrays must also be processed in parallel in an efficient way.



Bucket $B_{01} = \{t_1\}$ Bucket $B_{02} = \{\}$ Bucket $B_{03} = \{t'_1, t_4\}$

Tuples sent from P_0 to $P_1 = \{t_3\}$
Tuples sent from P_0 to $P_2 = \{t_2\}$
Tuples sent from P_0 to $P_3 = \{t'_2, t'_3, t'_4\}$
P_0's Local Meta Array for `count` aggregation $= \{0, 1, 0\}$
P_0's Global Meta Array for `count` aggregation $= \{0, 1, 2, 0\}$

Figure 11: Data distribution and meta arrays for P_0's local data

We propose to use a *local meta array* and a *global meta array* on each processor for tuples whose life spans overlap time-lines of multiple local buckets and multiple processors, respectively. Specifically, if the life span of a tuple t overlaps the k-th bucket ($B_{P_i k}$) of processor P_i through the l-th bucket ($B_{P_j l}$) of processor P_j, the tuple t will be replicated only in $B_{P_i k}$ and $B_{P_j l}$. Then, a local meta array of P_i is used to aggregate the information that the tuple t's life span overlaps the intermediate buckets $B_{P_i k+1}, \ldots, B_{P_i \mathcal{N}_B}$, and so is a local meta array of P_j for $B_{P_i 1}, \ldots, B_{P_i l-1}$. Finally, a global meta array is updated on a processor that owns the tuple t to inform the intermediate processors $P_{i+1}, \ldots, P_{j-1}$ of the existence of the tuple t overlapped with their time-lines. The size of a global meta array is equal to the number of processors. The i-th element of a global meta array stores an aggregate value (*e.g.*, `count`) for the i-th processor. Local meta arrays are identical with the ones used for the sequential bucket algorithm. Each processor computes its own global and local meta arrays independently.

In Figure 11, for example, suppose that tuples $t_1, \ldots, t_4$ are initially stored on a processor P_0, and four processors $P_0, \ldots, P_3$ participate in a `count` aggregation. Since the time interval of t_3 spans over three remote processors P_1, P_2 and P_3, t_3 is split into two segments t_3 and t'_3, which are then sent to the processors P_1 and P_3, respectively. Then, the third element of the global meta array of P_0 is incremented by one. In a similar way, t_4 is assigned to P_0's local bucket B_{03} and t'_4 is sent to processor P_3; the second and third elements of P_0's global meta array are incremented by one. Figure 11 shows the resulting data distribution across P_0's local buckets, data shipping to other processors, and P_0's local and global meta arrays. Note that there may be some tuples sent from other processors to P_0, but they are not shown in Figure 11.

The proposed parallel aggregation algorithm is summarized in Figure 12. In the algorithm description, it is assumed that the entire time-line of a table is partitioned into $\mathcal{N}_B \times \mathcal{P}$ disjoint intervals of an equal length, each of which is associated with a bucket, and the buckets are distributed across $\mathcal{P}$ processors by range partitioning so that each processor is assigned $\mathcal{N}_B$ consecutive buckets. This range partitioning scheme obviously minimizes the size of a global meta array in a way that only one array element is required per each processor. Since each processor computes a global meta array independently only for its local data, all the $\mathcal{P}$ processors need to communicate each other to compute a final global meta array for an entire database with respect to a given operator *op*. The operator *op* is determined by a kind of aggregate operation. For example, *op* will be an *addition* operator for a `count` aggregation and a *maximum* operator for a `max` aggregation. Such collective communication for computing a final global meta array can be implemented efficiently on most parallel computers and networks of workstations [1]. Thus the overhead for combining global meta arrays is expected to be negligible because the volume of communication is only $\mathcal{P}$ words per processor.

Algorithm 3 *Parallel Temporal Bucketization*

set $\mathcal{P} \leftarrow$ number of participating processors;
set $\mathcal{I}_B \leftarrow$ time interval for each bucket $((\mathcal{T}_{max} - \mathcal{T}_{min})/(\mathcal{N}_B \times \mathcal{P}))$;
set `this_proc` $\leftarrow$ a local processor id ($0 \leq$ `this_proc` $< \mathcal{P}$);
for each tuple `t` in a local partition or from a remote processor **do begin**
  set `start_proc` $\leftarrow$ (`t.start_time` $- \mathcal{T}_{min})/(\mathcal{I}_B \times \mathcal{P})$;
  set `end_proc` $\leftarrow$ (`t.end_time` $- \mathcal{T}_{min})/(\mathcal{I}_B \times \mathcal{P})$;
  if (`start_proc` $\neq$ `this_proc`) **then** send `t` to a processor P_{start_proc};
  if (`end_proc` $\neq$ `this_proc`) **then** send `t'` to a processor P_{end_proc};
  for (`i=start_proc+1` **to** `end_proc-1`) **do** update `global_meta_array[i]`;
  insert `t` into one or two local buckets as in **Algorithm 2**;
  update `local_meta_array` as in **Algorithm 2**;
end
Globally combine the `global_meta_array` wrt. an aggregate operator *op*;
for (`i=0` **to** $\mathcal{N}_B - 1$) **do begin**
  `local_meta_array[i]` $\leftarrow$ *op*(`local_meta_array[i]`,`global_meta_array[this_proc]`);
  perform temporal aggregation on the bucket B_i with `local_meta_array[i]` as in **Algorithm 2**;
end
end Algorithm

Figure 12: Parallel Bucket Algorithm based on Temporal Data Partitioning

6 Empirical Evaluation

In this section, we evaluate the proposed algorithms empirically and compare with the previous work. We chose a `count` temporal aggregation and carried out experiments under various operational conditions that may affect the performance of the algorithms. In particular, we focus on the performance gain by the proposed algorithms for small-scale aggregations, and the scalability of the sequential and parallel bucket algorithms.

6.1 Experimental Settings

Testing and benchmarks were performed on a cluster of 64 Intel Pentium workstations with 200 MHz clock rate. [2] Each workstation has 128 MBytes of memory and 2 or 4 GBytes of disk storage with Ultra-wide SCSI interface, and runs on Linux kernel version 2.0.30. The workstations are connected by a 100 Mbps switched Ethernet network. The switch can handle an aggregate bandwidth of 2.4 Gbps in an all-to-all type communication. For message passing between the Pentium workstations, we used the LAM implementation of the MPI communication standard [3]. With the LAM message passing package on the Pentium cluster, we observed an average communication latency of 790 microseconds and an average transfer rate of about 5 Mbytes/second. Note that this is relatively high latency and low transfer rate compared with parallel computers equipped with high performance switches such as IBM SP-2 parallel systems. [3]

For both sequential and parallel implementations, the same buffer size of 4 Kbytes was used for disk IO and message passing. Non-blocking message passing primitives were used in an attempt to minimize communication overhead by allowing inter-processor communication to be overlapped with local computation and disk IO. Throughout the experiments, we measured elapsed times including disk access time and communication overhead. For accurate measurement, we averaged elapsed times from multiple runs after eliminating extreme cases. Additionally, we avoided the system cache effects for disk accesses by loading

[2] For scalable performance evaluation, we were able to carry out experiments on only up to 32 workstations because several of them were under repair at the time of our experimental study.

[3] On a SP-2 system with a proprietary MPI implementation `mpif`, we observed an average communication latency of 55 microseconds and an average transfer rate of about 35 Mbytes/second.

irrelevant data into the entire memory between consecutive runs of our experiments.

We generated synthetic data in the same way as in [12]. Each database has a time-line of one million temporal instants. We considered two basic life spans for tuples: short-lived and long-lived. The life span of a short-lived tuple was determined randomly between one and 1,000 instants; the life span of a long-lived tuple was determined randomly between 200,000 and 800,000 instants, namely, between 20 and 80 percent of the time-line of a database. In most of our experiments, the population of long-lived tuples was fixed at 10 percent or 30 percent. The start times of tuples were uniformly distributed over the time-line of a database. Each tuple was 20 bytes including two temporal attributes (start time and end time) and other non-temporal attributes as well. Synthetically generated databases used in our experiments were not sorted by any temporal attribute unless stated otherwise.

6.2 Small-Scale Aggregation

The first set of experiments were carried out on relatively small databases between 1 MBytes and 20 MBytes so that all the required data structures can fit in available memory. Recall that the algorithms proposed in Section 3 as well as the aggregation tree algorithm and its variation require that the entire data structures be kept in memory. In this section, we used the *balanced tree* algorithm for `count` aggregations, and the *merge-sort aggregation* algorithm for `max` aggregations.

Figure 13(a) compares the balanced tree and aggregation tree algorithms for `count` aggregations; Figure 13(b) compares the merge-sort and aggregation tree algorithms for `max` aggregations. The proposed balanced tree and merge-sort aggregation algorithms consistently performed about twice faster than the aggregation tree algorithm for `count` and `max` aggregations, respectively. While the aggregation tree took more time to aggregate a database with higher percentage of long-lived tuples, the processing times of the two proposed algorithms remained constant for different percentage of long-lived tuples. Note that the performance of the aggregation tree algorithm remains unchanged for `count` and `max` aggregations, since the algorithm works essentially in the same way for both the aggregations.

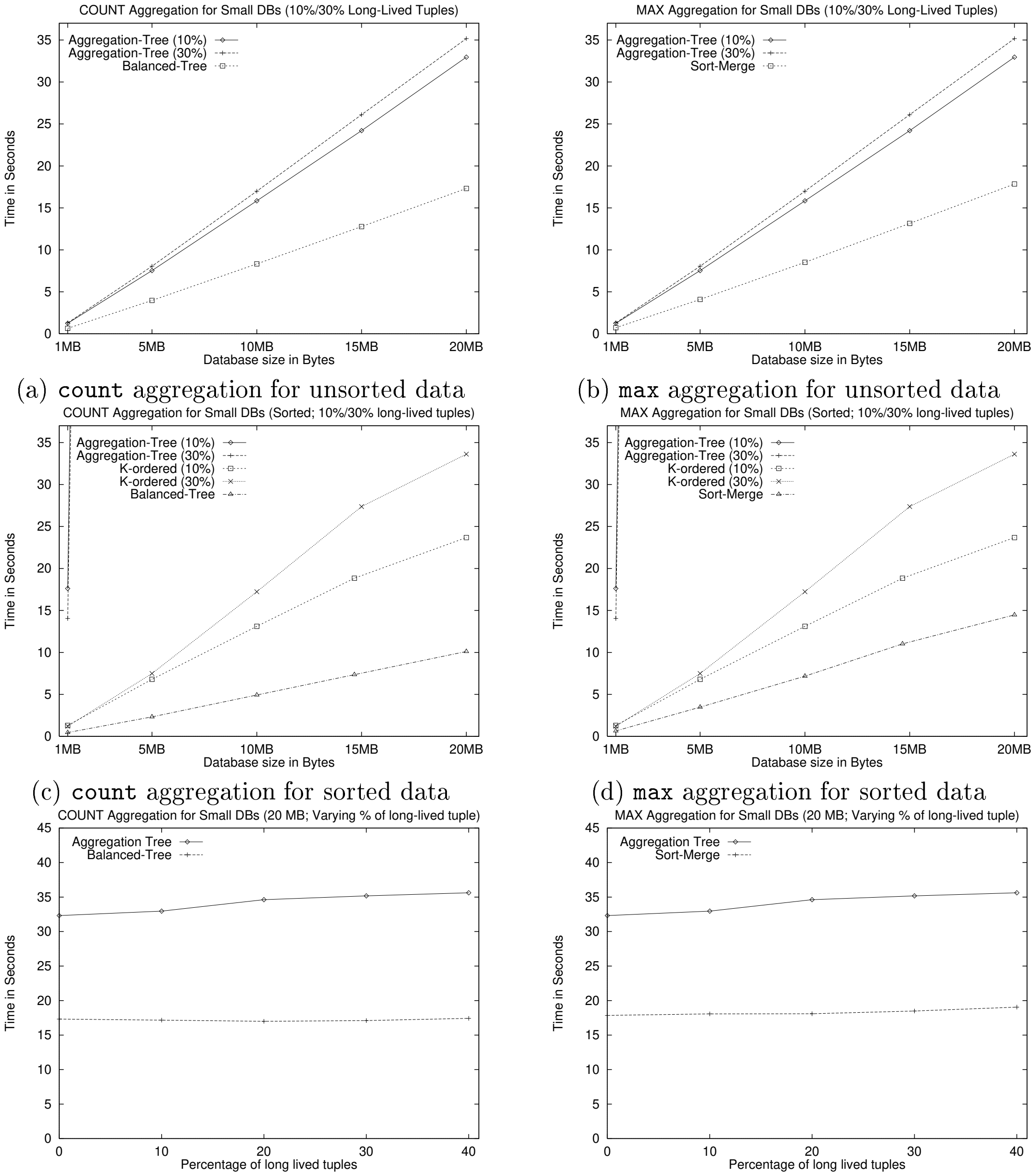
In Figure 13(c) and (d), the tuples in input databases were sorted by their start time, where we expected the worst-case performance from the aggregation tree algorithm. The processing times of the aggregation tree were several orders of magnitude slower than the two proposed algorithms, and were plotted as almost vertical lines in the figures. Thus, we compared with the k-ordered aggregation tree algorithm (with $k = 1$) instead. The proposed algorithm still performed two to three times faster than the k-ordered aggregation tree algorithm.

In summary, the proposed algorithms outperformed the aggregation tree and k-ordered aggregation tree consistently by a significant margin. The k-ordered aggregation tree requires a priori knowledge about the orderedness of databases, whereas the proposed algorithms do not. (Such knowledge will help reduce the processing time of the merge-sort algorithm, but it is not required.) The performance of the proposed algorithms was not affected by the percentage of long-lived tuples, as Figure 13(e) and (f) show the processing times measured on databases (20 MB) with a varying percentage of long-lived tuples.

6.3 Bucket Algorithm for Large-Scale Aggregation

Despite the fact that the balanced tree and merge-sort aggregation algorithms were designed for two different groups of aggregate operations, both algorithms showed almost identical performance behaviors in the previous experiments. Thus, for the rest of this section, we present experimental results only for `count` aggregations.

The second set of experiments were carried out to evaluate the *bucket* algorithm proposed in Section 4. First, we performed aggregations with and without data partitioning for small databases, so that we could measure the overhead of data partitioning. The balanced tree algorithm was used to compute `count` aggregates. In Figure 14(a), we used 64 buckets irrespective of database sizes, which was large enough to demonstrate the overhead of data partitioning. Compared with the balanced tree algorithm without data partitioning, we observed about 10 to 30 percent increase in processing time of the bucket algorithm. Despite the additional overhead, however, the bucket algorithm still outperformed the aggregation tree algorithm significantly. (Compare Figure 13(a) and Figure 14(a).)



(a) `count` aggregation for unsorted data

(b) `max` aggregation for unsorted data

(c) `count` aggregation for sorted data

(d) `max` aggregation for sorted data

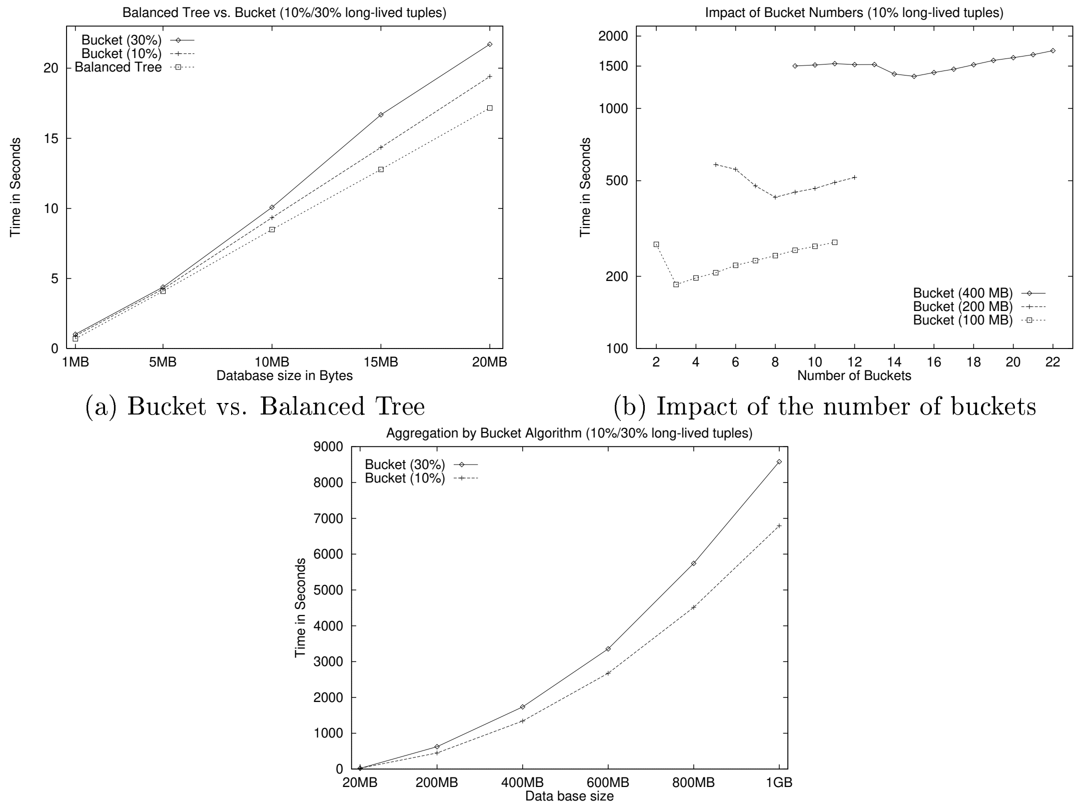
(e) `count` aggregation with varying long-lived

(f) `max` aggregation with varying long-lived

Figure 13: Aggregation time for small-scale databases

For small databases, the amount of overhead of data partitioning was expected to be smaller than what it should be for large databases, because all the buckets might remain in memory even after they were written to disk. So, for the next step of aggregating individual buckets, the cached buckets would be used instead of the disk copies. Also note that performance of the bucket algorithm is affected by the percentage of long-lived tuples. The reason appears quite obvious because long-lived tuples are more likely to be replicated than short-lived tuples, leading to increased computation time and disk access time.

From the experiments, we have noticed that performance of the bucket algorithm is affected by the number of buckets used for data partitioning. More interestingly, there seemed to exist local optimum



(a) Bucket vs. Balanced Tree

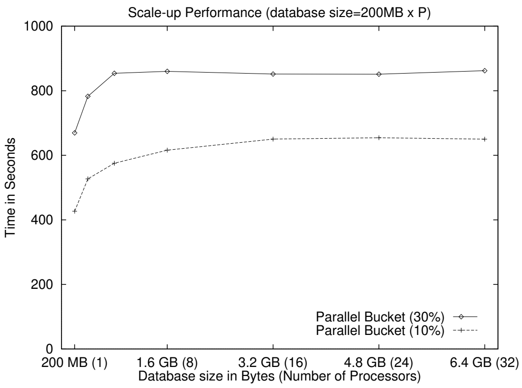
(b) Impact of the number of buckets

(c) Scaleup of Bucket algorithm

Figure 14: Aggregation time for large-scale databases

values, which were determined by database sizes. For example, in Figure 14(b), three, eight and fifteen were the optimal bucket numbers for a database of 100 MBytes, 200 MBytes and 400 MBytes with 10 percent of long-lived tuples, respectively. Our conjecture is that this is caused by two opposite performance effects from data partitioning. First, since the computational complexity of the balanced tree algorithm is higher than linear ($\mathcal{O}(N \log N)$), the overall computational complexity will be reduced by data partitioning. Specifically, the cost of a balanced tree construction is reduced from $\mathcal{O}(N \log N)$ down to $\mathcal{O}(N \log N - N \log \mathcal{N}_B)$, where N is the number of tuples and $\mathcal{N}_B$ is the number of buckets. Second, the more buckets are used for data partitioning, the more tuples are likely to be replicated, which will in turn increase the cost of disk accesses. We acknowledge that this issue should be addressed more carefully.

Figure 14(c) shows processing times of the bucket algorithm for databases of size from 20 MBytes up to 1 GBytes. The number of buckets used for data partitioning was 2, 8, 16, 24, 32 and 40 for 20 MBytes, 200 MBytes, 400 MBytes, 600 MBytes, 800 MBytes and 1 GBytes databases, respectively. Since each of these databases is too large to fit in memory (with an exception of a 20 MByte database), none of the small-scale aggregation algorithms could be used for this experiment. The results shown in Figure 14(c) demonstrate that the proposed bucket algorithm can compute temporal aggregates for databases substantially larger than the size of available memory. However, it should be noted that the processing time of the algorithm grows faster than linearly as the size of a database increases. This clearly motivates the need of scalable solutions such as the parallel bucket algorithm we proposed in Section 5.

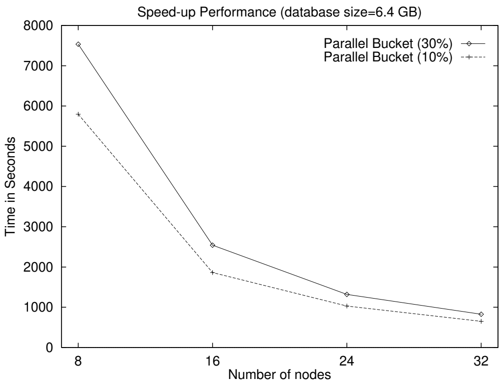


(a) Scale-up performance

$\mathcal{P}$	DB/$\mathcal{P}$ *MBytes*	Partitioning *Sec.* (%)	Aggregation *Sec.* (%)
1	200	71.0 (16.7)	355.9 (83.3)
2	200	127.8 (24.3)	399.3 (65.7)
4	200	176.3 (30.6)	399.2 (69.4)
8	200	207.2 (33.6)	408.7 (66.4)
16	200	196.2 (30.2)	454.2 (69.8)
24	200	183.5 (28.0)	470.9 (72.0)
32	200	185.4 (28.5)	464.7 (71.5)

(b) Separate measurements (10% long-lived)

Figure 15: Scale-up performance of parallel aggregation



(a) Speed-up performance

$\mathcal{P}$	DB/$\mathcal{P}$ *MBytes*	Partitioning *Sec.* (%)	Aggregation *Sec.* (%)
8	800	801.0 (13.8)	4995.2 (86.2)
16	400	373.3 (20.0)	1497.3 (80.0)
24	300	248.4 (24.1)	782.2 (75.9)
32	200	185.4 (28.5)	464.7 (71.5)

(b) Separate measurements (10% long-lived)

Figure 16: Speed-up performance of parallel aggregation

6.4 Parallel Algorithm for Large-Scale Aggregation

The third set of experiments were designed to evaluate the scalability of the parallel bucket algorithm proposed in Section 5. For all the experiments presented in this section, input databases were distributed across participating processors by round-robin partitioning on a non-temporal attribute. By choosing such a non-temporal partitioning scheme for initial data placement, we can effectively eliminate any potential advantage that the parallel bucket algorithm can exploit for better performance. On the other hand, range partitioning on a temporal attribute would be the most favorable data placement for the parallel bucket algorithm, because the number of tuples to be shipped to remote processors could be minimized and thereby reducing communication overhead.

For the scale-up performance measurements, we fixed the size of a database partition on each processor to 10 million tuples (*i.e.*, 200 MBytes), in a way that the entire database would grow proportionally as the number of processors increased. While the number of processors was varied from 1 to 32, the number of local buckets was fixed at 8. Thus, the total number of buckets used for data redistribution was $8 \times \mathcal{P}$, where $\mathcal{P}$ was the number of participating processors. The number of local buckets was determined from the previous experiments (see Figure 14(b)) based on the local partition size. Note that we used the sequential bucket algorithm for the case $\mathcal{P} = 1$.

In Figure 15(a), the scale-up plots were fairly close to a horizontal line, which indicated a nearly linear scale-up performance with respect to the increasing number of processors. This was corroborated by fact that the time spent on data partitioning remained quite static when the number of processors was no less than eight. See Figure 15(b) for measurements (from the case of 10% long-lived tuples) separated into two processing stages. As the number of processors was increased from one to two, data partitioning time was increased by about 80 percent, due mainly to additional cost for message passing between processors. In contrast, the time spent on aggregation was increased only by 12 percent due to increased data replication. As the number of processors increased, however, the increase of overhead leveled off and became essentially flat above the four processor case, and thereby allowing nearly linear scale-up performance.

For the speed-up performance measurements, we fixed the size of an entire database to 320 million tuples (*i.e.*, 6.4 GBytes), and determined the size of a database partition based on the number of participating processors. That is, the size of a local partition on a single processor was 6.4 GBytes/$\mathcal{P}$. Due to a limited disk space on each processor, we started experiments from 8 processors and increased the number of processors up to 32, changing the size of local database partitions accordingly from 800 MBytes to 200 MBytes. The resulting speed-up performance of the parallel bucket algorithm was shown in Figure 16(a).

As a matter of fact, it was surprising that a super-linear speed-up was observed whenever the number of processors increased. From the separate measurements in Figure 16(b) (from the case of 10% long-lived tuples), such a super-linear speed-up was largely attributed to the performance gain from local aggregation, which grew much faster than linearly as the number of processors increased Note that the number of buckets used for data redistribution increases proportionally to the number of processors. Thus, we conjecture that the overall aggregation cost is reduced by computing many smaller aggregations rather than computing a few larger aggregations.

7 Conclusions and Future Work

We have developed new algorithms for computing temporal aggregates. The proposed algorithms provide significant benefits over the current state of the art in different ways. The balanced tree and merge-sort aggregation algorithms have improved the worst-case and average-case processing time significantly for small databases that fit in memory. We have also developed new sequential and parallel bucket algorithms based on novel data partitioning schemes. These algorithms can be used to compute temporal aggregates for databases that are substantially larger than the size of available memory, by processing data partitions in a sequential or parallel fashion. In particular, with the local and global meta arrays for partitioned data, we have demonstrated that the parallel bucket algorithm achieves scalable performance for large-scale databases by delivering nearly linear scale-up and speed-up.

From our experiments, we have observed that there are a few factors that affect the performance. They include the percentage of long-lived tuples and the number of buckets used for data partitioning. Although the proposed algorithms outperformed previous approaches consistently irrespective of such conditions, we believe it is worth elaborating further on the issues. In this paper, we assumed that tuples were uniformly distributed within a time-line of a database. The performance of the proposed solutions may degenerate if there exist skews in data distribution. We will investigate the use of adaptive data partitioning or data sampling techniques to handle such data skews. Additionally, we plan to study performance impacts of such factors as initial data placement (*e.g.*, temporal partitioning vs. non-temporal partitioning) and data reduction by aggregation.

We also plan to extend the data partitioning approach to spatio-temporal databases, which requires computing aggregates for data objects with two or more dimensional extents. Unlike the temporal aggregation, we expect that the process of data partitioning and generating meta arrays will be more sophisticated.

References

[1] M. Barnett, S. Gupta, D. Payne, L. Shuler R. van de Geijn, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 357–364, Knoxville, TN, May 1994.

[2] Jon Louis Bentley. Algorithms for Klee's rectangle problems. Technical Report unpublished, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1977.

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

[4] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1), March 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, Cambridge, Mass., 1990.

[6] Anindya Datta, Bongki Moon, and Helen Thomas. A case for parallelism in data warehousing and OLAP. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications (DEXA'98)*, pages 226–231, Vienna, Austria, August 1998.

[7] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 1–8, Boston, MA, June 1984.

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

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

[10] Jose Alvin G. Gendrano, Bruce C. Huang, Jim M. Rodrigue, Bongki Moon, and Richard T. Snodgrass. Parallel algorithms for computing temporal aggregates. To appear in Proceedings of the 15th International Conference on Data Engineering (ICDE'99). Also available as Technical Report TR 98-9, University of Arizona, Department of Computer Science.

[11] Christian S. Jensen and Richard T. Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, 1996.

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

[13] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Reading, Mass., 1973.

[14] R. T. Snodgrass et al. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

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

[16] 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.

[17] A. Tansel et al. *Temporal Databases: Theory, Design and Implementation.* Database Systems and Applications Series. Benjamin/Cummings, Redwood City, CA, 1993.

[18] Transaction Processing Performance Council, San Jose, CA. *TPC Benchmark D (Decision Support) Standard Specification*, revision 1.2.3 edition, June 1997.

[19] P. A. Tuma. Implementing historical aggregates in TempIS. Master's thesis, Wayne State University, Detroit, Michigan, November 1992.

[20] Carolyn Turbyfill, Cyril Orji, and Dina Bitton. AS^{3}AP: An ANSI SQL standard scaleable and portable benchmark for relational database systems. In Jim Gray, editor, *The Benchmark Handbook : for Database and Transaction Processing Systems (2ed)*, pages 317–357. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1993.

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