Computing Temporal Aggregates

Nick Kline & Richard T. Snodgrass
Department of Computer Science
University of Arizona
Tucson, AZ 85721
kline, rts@cs.arizona.edu

Abstract

Aggregate computation, such as selecting the minimum attribute value of a relation, is expensive, especially in a temporal database. We describe the basic techniques behind computing aggregates in conventional databases and show that these techniques are not efficient when applied to temporal databases. We examine the problem of computing constant intervals (intervals of time for which the aggregate value is constant) used for temporal grouping. We introduce two new algorithms for computing temporal aggregates: the aggregation tree and the k-ordered aggregation tree. An empirical comparison demonstrates that the choice of algorithm depends in part on the amount of memory available, the number of tuples in the underlying relation, and the degree to which the tuples are ordered. This study shows that the simplest strategy is to first sort the underlying relation, then apply the k-ordered aggregation tree algorithm with $k = 1$.

KEYWORDS: Temporal Databases, Aggregate Computation, Query Evaluation, Query Optimization

1 Introduction

Aggregate functions are evaluated on relations and compute a scalar value, such as the average salary of all employees. Aggregate functions are an important component of data query languages, and are heavily used in many applications. Query benchmarks often contain a large percentage of aggregate queries (e.g., [Gray 1991]). Hence, efficient execution of aggregate functions is an important goal.

In temporal databases, relations model time-varying aspects of an enterprise. Information such as when the tuple was written to disk (known as *transaction time*), or when the tuple was known to be valid (known as *valid time*) may be represented [Jensen et al. 1994]. Temporal database models and query languages have recently been developed that require new implementation strategies for aggregate functions [Kline 1993, Tansel et al. 1993].

One reason that existing approaches are not efficient is due to *temporal grouping*, where we may wish to group the results by time. In this paper we focus on aggregates computed over interval relations grouped by instant, where we wish to know the aggregate value at each point in time. This is perhaps the most common grouping. This computation is difficult 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.

In this paper we present efficient implementation techniques for temporal aggregates. We describe the query language constructs used to express aggregates in snapshot and temporal query languages. We analyze related work on the evaluation of snapshot aggregates and consider their efficacy for temporal aggregate computation. We introduce two different algorithms and implementation strategies more suitable for temporal databases: the aggregation tree and the k-ordered aggregation tree. We describe how to implement these algorithms assuming sufficient memory is available. We evaluate the performance of these algorithms by evaluating queries over relations differing in size, number of long-lived tuples, and the degree to which the relation is sorted.

We empirically examine both the space and time requirements of the algorithms and specify how a query analyzer would choose the proper technique. Example temporal queries will be specified in TSQL2 [Snodgrass et al. 1994]. TSQL2 extends SQL-2 in an upward compatible manner to support many aspects of temporal databases, including temporal aggregation.

2 Aggregates in Query Languages

Aggregates in most relational query languages may be divided into two types, *scalar aggregates* and *aggregate functions*. Scalar aggregates yield single scalar values, while aggregate functions return relations.

For example, the scalar aggregate in the SQL query

```
SELECT AVG(Salary)
FROM Employed
```

will compute the average salary of all the employees and return a scaler value.

Aggregate functions may return a set of values because of qualifications in the query. The tuples being aggregated must be partitioned as specified in the group-by clause. The following query will compute the average salary of employees grouped by department, and return these values as a new relation.

```
SELECT Dept, AVG(Salary)
FROM Employed
GROUP BY Dept
```

Scalar aggregates may be computed and then replaced by their value in their query, since they are independent of the query in which they are nested [Epstein 1979].

In temporal databases, we extend the domain and range of aggregates to include time. In this paper, we assume that the temporal dimensions are intervals; aggregates may also be evaluated over event relations. We also extend the group-by clause with temporal grouping. We will term the beginning time of an interval the *start* time and the terminating time the *end* time.

Query language features to support aggregation are included in TSQL2 [Kline et al. 1994]. Aggregates in databases either select their values (such as the aggregate maximum) or compute their values (such as the aggregate count). In TSQL2, aggregates are defined over both temporal and non-temporal values. If the previous query were computed over a temporal relation, then the result returned would still be the average salary grouped by department, but this would be a time-varying value. The average per department would vary over time reflecting the information in the database changing over time.

Besides new aggregates, TSQL2 provides extended query language features to support aggregation. One new feature is *temporal grouping.* Temporal grouping is the process in temporal databases where the time-line is partitioned over time, tuples are grouped over these partitions, and aggregate values are computed over these groups. There are two types of partitioning, by a *span* (a calendar defined length of time, such as a year), or by each instant (an instant is the smallest measurable period of time in a temporal database). In this paper we consider partitioning by instant.

For partitioning by instant, we first create a partition of the underlying relation for each instant. We then compute the aggregate over each partition. Suppose that for two consecutive instants, the same tuples overlap both of them. The aggregate will have the same value for both of these partitions, since they both are computed over identical sets of tuples. This will be true for any sequence of instants for which the same tuples overlap. We call these sequences of instants (or partitions) where the aggregate values do not change (and equivalently the group of tuples they are computed over do not change) *constant intervals.* Notice that although we describe a partitioning of a time-line, the partitioning is determined by the tuples.

3 Snapshot Aggregate Computation

Aggregate computation in conventional databases is well-understood. In this section we will describe the typically used techniques used to compute aggregates in snapshot databases and also discuss several optimizations.

A scalar aggregate is composed of an aggregate expression and an optional qualification. Epstein outlined a simple algorithm for evaluating scalar aggregates consisting of two steps [Epstein 1979].

1. Allocate a tuple to hold the result. This tuple contains two attributes, a counter (initialized to zero) used to count the number of tuples that satisfy this aggregate's qualification, and a result attribute.
2. For each tuple that qualifies, update the counter and the aggregate result.

The count field is used for computing aggregates that need to know how many tuples satisfied the qualification, such as count and average. For other aggregates, such as minimum and maximum, it may be used to recognize the first tuple.

To handle many scalar aggregates in a query, compute each of them separately and store each result in a singleton relation, referring to that singleton relation when evaluating the rest of the query.

4 Temporal Aggregate Computation

In this discussion, we first consider extending existing approaches to aggregate computation. The problem we address is how to compute a temporal aggregate over intervals of a time-line. These intervals are constant intervals, induced by the timestamps of the underlying relation. Then we discuss some general points about computation of constant intervals.

4.1 A Previous Implementation

One approach for implementing aggregation in a temporal query language is based on an extension of existing approaches [Tuma 1992]. This approach supports temporal aggregates using extensions of existing aggregation techniques but, as stated in the referenced paper, the efficiency suffers. Basically, the constant intervals are determined first, then the aggregate is evaluated using the technique described above. Specifically, five steps are involved.

First, determine the periods of time during which the relation remained fixed. These are the times during which no new tuples entered or exited the relation and hence are constant intervals. For each constant interval, select the tuples which overlap it. Third, if there is a group-by clause present, partition each constant interval of tuples into subsets, where each subset has a different unique value for the partitioning attribute. These are referred to as *aggregation sets.* Fourth, compute the aggregate value for each aggregation set. Finally, associate these values with the proper combination of tuples from the original query, based on the values indicated by the group-by clause, time interval of the aggregation set, and interval or event from the valid clause in the original query.

Since the computation of constant intervals is computed first, and then the aggregate values are computed for each constant interval, the relation must be read twice. The algorithms we present below need only to read the relation once.

4.2 Extending Epstein's Approach

We may extend the temporary relation approach of Epstein (used to handle group-by clauses) to manage the constant intervals for aggregation in temporal databases. We do this by replacing or supplementing as appropriate the *group-by-value* with an *interval-value* which represents the interval over which we are computing the aggregate. Each element's interval represents a constant interval. To compute the constant intervals (and the aggregate value over each at the same time), we use a temporary relation to maintain a list of the constant intervals and their aggregate values, incrementally updating this list for each tuple. This may also be formulated as an optimization of the algorithm from the previous section, where we have combined the computation of the aggregate (step 4) with the computation of the constant intervals (steps 1–3).

Implementing this simple extension to Tuma and Epstein's work requires considering each tuple only once. When we are ready to consider a new temporal tuple, we simply compare the tuple's start and end times with the start and end times of each interval in the list. If the tuple's interval overlaps a list element's interval, then we update the element's aggregate value. We will implement this algorithm using a linked list and call this the naive or linked list approach.

5 New Algorithms

In this section we introduce several new algorithms which may be used to compute a temporal aggregate. For these algorithms, we assume that there is sufficient main memory to store the structures.

First we will discuss an example temporal aggregate query. We use 0 as the origin or earliest timestamp and ∞ as the greatest timestamp.

Figure 1 shows an example temporal relation. This relation maintains the period of time that people were employed by a company. Notice that "Nathan" was not employed during times [13, 17], and that the relation is in no particular order. We assume that the intervals are closed intervals.

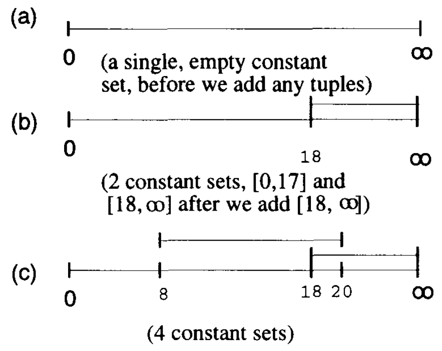
name	*salary*	*start*	*end*
Richard	40K	18	∞
Karen	45K	8	20
Nathan	35K	7	12
Nathan	37K	18	21

Figure 1: The Employed Relation

In Figure 2 we see how the Employed relation induces constant intervals. The tuples are shown above the time-line. In Figure 2.a, we have a timeline with a single constant interval. In Figure 2.b, we see the constant intervals induced by timestamps of the first tuple [Richard, 40K, 18, ∞]. Since only the 18 is a unique timestamp we only add one constant interval. In Figure 2.c, we see that adding a tuple with two unique timestamps adds two new constant intervals. Each unique timestamp adds one more constant interval. So with 6 unique timestamps and the initial constant interval, we have 7 constant intervals induced by the 4 tuples in the Employed relation.

5.1 The Aggregation Tree

In this section we introduce the *aggregation tree* algorithm. We describe how to incrementally construct a tree structure which manages the constant intervals and computes the aggregate values. We describe an



(a)
0
(a single, empty constant set, before we add any tuples)
∞
(b)
0
18
∞
(2 constant sets, [0,17] and [18,∞] after we add [18, ∞])
(c)
0
8
18 20
∞
(4 constant sets)

Figure 2: Constant Intervals Induced by The Employed Relation

evaluation technique which assumes sufficient memory is available for construction of the tree. There are other techniques which may be used to implement the aggregation tree with only limited memory resources, such as preallocating the tree in a linear memory array, thus avoiding the need for tree node pointers, but we will not discuss these alternatives here.

The following TSQL2 query will compute the number of tuples valid over each constant interval.

```
SELECT COUNT(Name)
FROM Employed E
```

The default grouping expression in TSQL2 groups queries by instant, which means we will compute the aggregate value separately at each point in time. The result is coalesced by valid-time such that each interval in the result is a constant interval with at least one instant.

Recall that the Employed relation records the times that certain people were employed. The result of this query when applied to the Employed relation is shown in Table 1.

count	*start*	*end*
0	0	6
1	7	7
2	8	12
1	13	17
3	18	20
2	21	21
1	22	∞

Table 1: Result of Temporal Aggregate Query

The algorithm proceeds in two steps: build the aggregation tree, and perform a depth first search to compute the aggregate values at the leaves. Each node in the tree has an aggregate state value (here it is a count of the number of nodes which overlap this constant interval) and a start and end time. The start and end times at a leaf node encode a constant interval in the query's result. Each leaf also contains the partial aggregate result for that constant interval. Initially, a single node valid from 0 to ∞ has a count of 0 (Figure 3.a). For each tuple, we search for the constant interval(s) containing the start and end times. When we find these intervals, if the timestamp falls between the boundaries of the constant interval, we split the interval in two. Figure 3.b shows the effect of adding the first tuple's interval, [18, ∞]. First we search in the initial tree(Figure 3.a) for the bounding interval for 18. We see that the ending time of the tuple's interval (∞) was the same as the ending time of the tree node's interval. So, we split the node into two pieces. We do not need to search further for the tuple's ending timestamp (∞) because that was contained in the node.

The algorithm continues by considering the second tuple, with a valid timestamp interval of [8, 20]. The start time, 8, is searched for in the current tree, Figure 3.b. The bounding node is [0, 17] but the constant interval extends past the end of this node so the searching must continue. The node [0, 17] is split according to the tuple's start time of 8. Searching continues for the ending timestamp, 20, and again a node is split. The result of processing this second tuple is Figure 3.c. Notice that only the count values stored at the leaves were adjusted. The node [8, 17] has a count of 1 because this is the part of the previous constant interval [0, 17] which is overlapped by the current tuple. The other child of [0, 17], [0, 7], is not overlapped by the current tuple so its aggregate value is initialized to 0. We adjust the internal node aggregate values when a tuple's constant interval completely overlaps a node. We continue processing the tuples and the final result is shown in Figure 3.d.

One advantage of this algorithm is that it is not always necessary to search the leaf nodes of the tree. Suppose that to the final tree, we wished to add a tuple with a constant interval of [5, 50]. We would search the tree for the constant interval containing 5. We find the node [0, 7] and split it. We continue search the tree, and see that the node [8, 17] is completely overlapped by our tuple interval. We update the aggregate value stored here, the count, to 2. We need to do this for the intervening nodes in the tree. But, since we completely overlapped node [8, 17], we did not need to search the tree past this node to its leaves, we only needed to update the value stored at this node. We would continue processing this new tuple until we found the constant interval which overlaps the end timestamp, 50.

After the tree is completed, a depth first search is performed from the root, keeping track of the aggregate additive count value as we recurse. This will produce the result in time order. Whenever we reach a leaf node we write the aggregate value out with the constant interval. For example, when we reach leaf node [8, 12] (in the final aggregation tree, Figure 3.d), we add the aggregate value of the leaf node's parents (which is $0 + 0 + 1$) to the leaf's value of 1 and get 2.

Since our tuples are not sorted, a tuple may be inserted in any part of the tree. This could be inefficient if we have large numbers of tuples and limited memory, as we could have a large working set of memory pages. The aggregation tree works best if the relation is randomly ordered by time, since the tree that

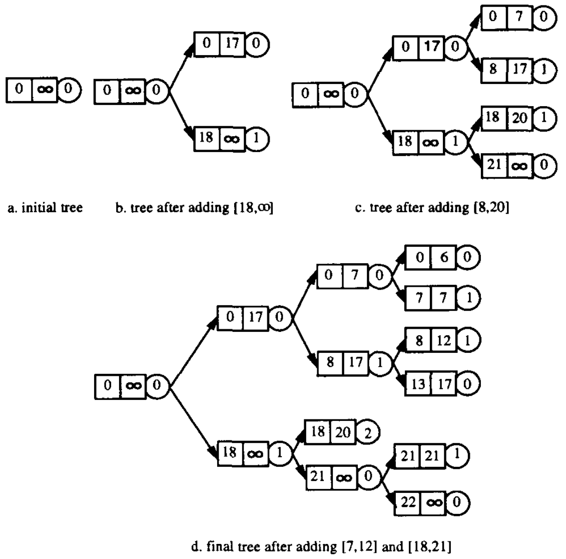

a. initial tree
b. tree after adding [18,∞]
c. tree after adding [8,20]
d. final tree after adding [7,12] and [18,21]

Figure 3: Aggregate Computation Tree

results is more balanced.

The aggregation-tree is similar to the *segment tree* [Preparata & Shamos 1985]. A segment tree is a balanced structure used to store segments of numbers from the real number line. The use of segment-like trees to store constant intervals for aggregate computations is one of the contributions of this paper.

The worst case time to create the tree is $O(n^2)$ because, in the worst case, the tuples are ordered in time, and the tree becomes a linear list. In Section 5.3 we propose a variation of the aggregation tree for sorted and almost sorted relations.

If we do not balance the aggregation tree, then it is simple to page portions of the tree to disk. This is relatively easy because it is simple to mark a parent as pointing to a subtree not currently in memory. Simply accumulate the tuples which would overlap this region of the tree and process them later. This should be an interesting area for future research.

5.2 Data Considerations

In this section we describe several ways to quantify the sortedness of a relation. We then describe several algorithms which exploit the sortedness of the relation.

We define *totally ordered* by time to mean that the tuples are sorted in order by start-times (typically in increasing order), with ties broken by using the end time. Notice that this definition does not consider the degree to which the tuple intervals overlap other tuples.

Another quantification is how far from being totally ordered a group of tuples is. We call a set *k-ordered* if each tuple is at most *k* positions from its position in a totally ordered version of the relation. A totally ordered set of tuples is equivalently 0-ordered. One way which this might arise is if all tuples written to a database are written around the time they are known to be true. For example, if a programmer was hired on Tuesday, we probably write her new salary information to the database on Tuesday or Wednesday. If we guaranteed that we always wrote the information by the next day, then we would have a form of temporal specialization, specifically, a retroactively bounded relation, which is common in practice [Jensen & Snodgrass 1994]. A more efficient variant of the aggregation tree may be applied to both k-ordered relations and retroactively bounded relations (discussed in Section 5.3).

Another quantification is the characterization of how many tuples are out of order, and how far they are from their totally sorted position. If there are n tuples in a relation, and the tuples are k-ordered, then we define the *k-ordered percentage* as the following quotient,

$$k\text{-ordered-precentage} = \frac{\sum_{i=1}^{k} i * n_i}{k * n},$$

where n_i is the number of tuples i positions out of order. This ratio ranges from 0 to 1. If the tuples are all in order, then this ratio will be 0. If the tuples are maximally disordered by k, then the ratio is greater than 0; the higher the disorder, the higher the ratio. The ratio can be 1 only for certain k. For a relation with 6 tuples, with $k = 3$, if we swap tuples 1 with 4, 2 with 5, and 3 with 6, we have a k-ordered-precentage of 1 ($= (3 + 3 + 3)/(3 * 3)$).

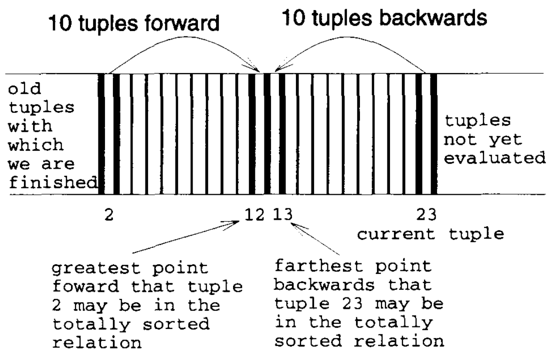
Several example k-ordered-percentages follow in Table 2. These examples illustrate how different ordering affects the value of the k-ordered-percentage.

k-ordered percentage	*Explanation*
0	the tuples are sorted
0.0002	2 tuples 100 places apart are swapped
0.002	20 tuples are 100 places from being sorted
0.00505	1000 are 50 places out of order
0.0505	10 tuples are 1 place out of order, 10 are 2, ..., 10 are 100
0.1	1000 are 100 places out of order
0.505	100 are 1 place out of order, 100 are 2, ..., 100 are 100 out of order; the other tuples are in order

Table 2: Examples of k-ordered-percentages (n=10000, k=100)

5.3 Garbage Collecting the Aggregation Tree

If the tuples in a relation are k-ordered, then we may garbage collect the left side of the aggregation tree as we create it. We may remove some nodes from the beginning of the tree because we can determine when we have considered all tuples which affect their values. Before we remove these nodes, we send their associated intervals and aggregate values to the next stage of query evaluation. We call these variations of the aggregation tree *k-ordered aggregation trees*, because they depend on the properties of k-ordered relations.



10 tuples forward
10 tuples backwards
old tuples with which we are finished
tuples not yet evaluated
2
12
13
23
current tuple
greatest point foward that tuple 2 may be in the totally sorted relation
farthest point backwards that tuple 23 may be in the totally sorted relation

Figure 4: k-ordered list for $k = 10$

For illustrative purposes, let us suppose k is 10. The algorithm begins processing the tuples by building a typical aggregation tree with the first 22 tuples ($2k + 2$), as described in Section 5.1. As the algorithm proceeds through the tuples, it is necessary to keep the last $2k + 1$ (21 in this case) tuple intervals in a list. Begin numbering with 1 for convenience. Now, consider adding the next tuple, the tuple at position 23. Tuple number 2 could have been at most 10 (or k) positions out of order. Thus, tuple number 2 could be placed, at the greatest, at position 12 (or $2 + k$, which is the original position plus 10 possible movement = 12) in the totally ordered list of tuples. The current tuple (number 23) could have been, at the earliest position in the list, at position 13 ($23 - 10 = 13$). Figure 4 shows this relationship. Since tuple 23 must come after tuple 2 in the totally ordered list, and any tuples after 23 would also appear after tuple 2 in the totally ordered list, then it follows that the algorithm is finished with any constant intervals whose end time is before the start of tuple number 2.

What this means for the processing of a k-ordered relation is that after we process each tuple, we look back at the tuple $2k+1$ (21 back in our example) back from the current tuple. The worst case running time of the algorithm is still $O(n^2)$, but we have reduced the main memory space requirement substantially.

We garbage collect the nodes by keeping track of two pieces of information as we build the tree. As described above, we keep a window of the last $2k + 1$ tuple timestamps to use in determining when we can garbage collect a piece of the tree. The second piece of information we maintain is the *current earliest* constant interval in the tree still remaining in the tree.

The garbage collection proceeds as follows. After processing a node, the tuple timestamp $2k + 1$ nodes back in the relation is examined; any node may be

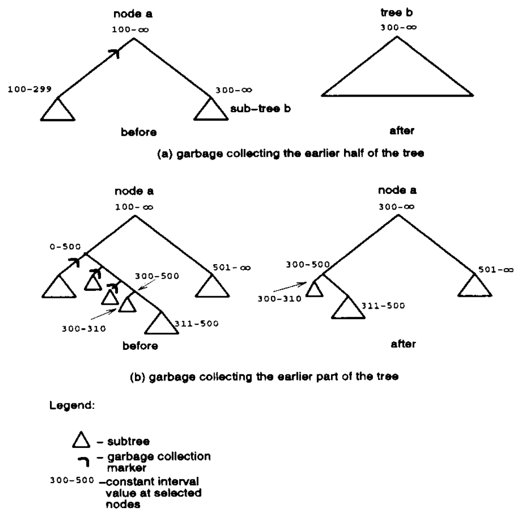

node a
100-∞
100-299
300-∞
sub-tree b
before
tree b
300-∞
after
(a) garbage collecting the earlier half of the tree
node a
100-∞
0-500
300-500
501-∞
300-310
311-500
before
node a
300-∞
300-500
501-∞
300-310
311-500
after
(b) garbage collecting the earlier part of the tree
Legend:
△ - subtree
- garbage collection marker
300-500 -constant interval value at selected nodes

Figure 5: Garbage Collecting Before Time 300

garbage collected whose associated constant interval ends before this previous tuple's start time (call this start time the *gc-threshold*). Examine the time stored at the root of the aggregation tree. If the left half of the tree occurs before the gc-threshold, the entire left subtree may be garbage collected, and the root removed, and the root may be replaceed with the root's right child (or later child). This is illustrated in Figure 5.a. If it is not possible to remove the root's left child, compare the gc-threshold to the current earliest constant interval. If the earliest constant interval is before the gc-threshold, then the algorithm may garbage collect some interval or intervals in the earlier half of the aggregation tree, as shown in Figure 5.b. In this case, if only the earlier of two leaves of a node are garbage collected, the parent is removed and replaced with the remaining leaf. Since the algorithm is only garbage collecting the earliest consecutive part of a tree, a "hole" is never created in the constant intervals. Testing continues until these two conditions for garbage collection are both false.

6 Empirical Comparison

We implemented the algorithms discussed above to evaluate the effects of memory usage and different relation ordering. Table 3 provides the parameters used in our testing to contrast the different algorithms. We performed the tests on a lightly loaded Sun IPC SPARCstation running SunOS 4.1.1, and the tests were compiled using `gcc2` with optimization turned on.

We utilized a test relation with a tuple size of 128 bytes, which contained four germane attributes: *name* (6 bytes), *salary* (4 bytes), *start-time* (4 bytes), *stop-time* (4 bytes), as well as attributes not examined by the aggregate (110 bytes). We utilized 4 byte timestamps since this was sufficiently large for our relation's lifespan. TSQL2 permits the range and granularity of the timestamps to affect the allocated size of timestamps; we expect one word timestamps to be common [Dyreson & Snodgrass 1994].

All of the algorithms tested read the relation only one time, so we did not measure disk access time. Note that this is in contrast to Tuma's explicit constant interval algorithm, which is the only temporal aggregate algorithm implemented prior to our work. Tuma's implementation required that the underlying relation be scanned twice, once to compute the constant intervals, and again to compute the aggregate over each interval [Tuma 1992]. We tested different relation sizes by creating relations from 128K to 8M in size (1K to 64K tuples), doubling the relation size between tests. We did not test larger relation sizes because the sizes we used are sufficient to characterize the differences in algorithm performance.

Parameter	*Values tested*
k	4, 40, 400
k-ordered-percentage	0.02, 0.08, 0.14
Long-lived tuples	0%, 40%, 80%
Size of the relation in tuples	1K, 2K, 4K, 8K, 16K, 32K, 64K
Size of the relation in bytes	128K, 256K, 512K, 1M, 2M, 4M, 8M

Table 3: Test parameters

We found that the choice of aggregate did not materially alter the results. We thus provide results only for the count aggregate. Count uses only 4 bytes per each aggregate-value stored. The other aggregates would require more memory if they were tested. Sum, maximum, and minimum all use 4 bytes, plus an additional bit to mark an empty value. Average uses 8 bytes, 4 for the sum and 4 for the count. This information is in addition to the pointers and timestamp values, which account for a large portion of the main memory requirements.

Our relation had a lifespan of 1 million instants. We generated the starting position of our tuples independently, so our relations had many unique timestamps. Realistic data would likely have a smaller percentage of unique timestamps, with an associated increase in performance for the tree based algorithms. We consider two basic tuple lifespans. First, short-lived lifespan tuples are tuples whose lifespan is a random length from 1 to 1000 instants. Second, long-lived lifespan tuples have duration equal to a random length between 20% and 80% of the relation's lifespan (200,000 to 800,000 instants). Generated tuples that extend past beyond the relation's lifespan were discarded. We tested different amounts of long-lived tuples on the algorithms. The presence of long-lived tuples severely affected some algorithms.

We ran each test several times with different random number seeds to establish reliable results. We do not show the error bars since 95% confidence intervals never exceeded 10% of the indicated value on any of the tests, an especially small figure on graphs with logarithmic axes.

We first tested the linked list algorithm, and the aggregation tree algorithm with randomly ordered relations, using varying relation sizes and varying percentages of long-lived tuples as given in Table 3. We indicate only CPU time, as all algorithms perform a single segmented scan of the input relation.

We then added two parameters to relation generation. We generated a sorted relation, and then altered it according to various k-ordered and k-ordered-percentages. While it is perhaps more realistic to test on retroactively bounded relations ([Jensen & Snodgrass 1994]), such as updates occuring within two days, modeling such relations is more difficult. So instead, we approximate a retroactively bounded relation with a k-ordered relation. For a uniform arrival rate, the two are identical.

As before, we also tested different relation sizes and percentages of long-lived tuples. For these tests, we compared the running times of the linked list algorithm and the k-ordered aggregation tree.

Finally, we tested the algorithms on sorted relations. We compared the linked list algorithm, the aggregation tree, and the k-ordered aggregation tree with $k = 1$.

The results are presented in the following sections. Please be sure to note that the results are *log-log* graphs, and so the results may appear to be deceptively close. We use logarithmic graphs since we increased the relation size by a factor of two for each test.

6.1 Query Evaluation Time

In Figure 6 we see that the query evaluation time for randomly ordered relations depends on the relation size and the percentage of long-lived tuples. At current processing and I/O rates, all of these algorithms will be compute bound at realistic relation sizes. We provide results for computation time only. Since the performance of the aggregation tree and the linked list was unaffected by the presence of long-lived tuples, we provide only a single result for each. The linked list had the worst performance over all relation sizes. For the largest relation, it was 300 times slower than the aggregation tree.

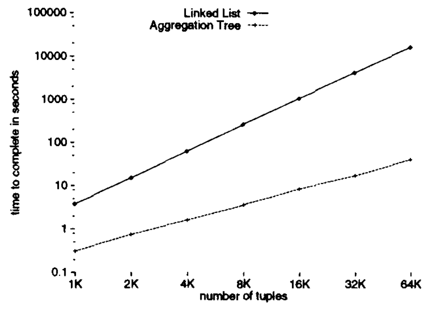


Figure 6: Time Comparison on Unordered Relations

Next we consider ordered and partially ordered relations. We altered the order of a sorted relation according to various k and k-ordered-percentages test values. We see in Figure 7 how the various k values affect the results for no long-lived tuples (in the legend, entries *Ktree K=4*, *Ktree K=40*, *Ktree K=400*). We combined these results with the results of computing the algorithm over totally ordered relations for comparison purposes. The effect of the k-ordered-percentage was outweighted greatly by the effect of the k value (especially on a log-log graph), so we only show a single graph for each k value; basically, larger k-ordered-percentages meant a more random tree which lead to a small increase in performance. Smaller values of k resulted in more efficient run-time performance for the k-ordered aggregation tree algorithm. This is because smaller k values mean a smaller "history" must be maintained.

Figure 7 also contains test results for the linked-list, the aggregation tree, and the k-ordered aggregation tree with $k = 1$, all when computed over ordered relations with no long-lived tuples. The linked list algorithm was relatively unaffected by the new parameters tested here. As discussed above, when the tuples are sorted or nearly so, the standard aggregation tree has performance near $O(n^2)$, and obviously the performance suffers in this test case. For this test case, we sorted the relation before applying the aggregation tree algorithm.

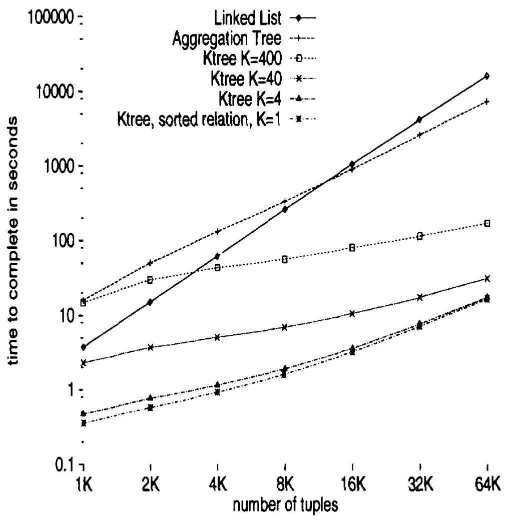


Figure 7: Time Comparison on Ordered Relations without Long-lived Tuples

In Figure 8, we see how the algorithms are affected by the presence of many long-lived tuples. As in the previous figure, the k-ordered aggregation tree algorithms are computed over *partially ordered* relations; the other algorithms are computed over ordered relations. Most algorithms are slowed here since they need a larger state, as discussed below. The linked-list is unaffected; the aggregation tree suffers from the presence of long-lived tuples. If the relation is ordered, we

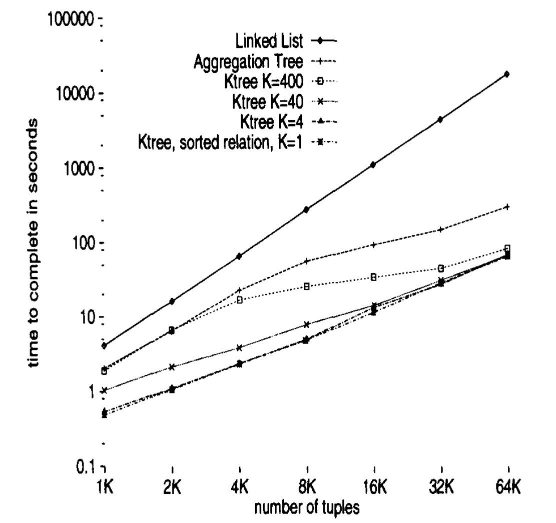


Figure 8: Time Comparison on Ordered Relations with 80% Long-lived Tuples

may use the k-ordered aggregation tree with a window of 1 as we tested here. This results in a very efficient run-time performance, and as we shall see, efficient memory utilization. As discussed before, the aggregation tree has $O(n^2)$ performance over sorted relations.

The behavior of the k-ordered aggregation tree appears to be complex. However, the results can be summarized as follows. Small values of k are more efficient because the number of tuples that are maintained in the tree is smaller. Larger values of k slow processing down, because a sorted or almost sorted relation leads to creation of a linked list. Large k-ordered-percentages improve the run time performance of the k-ordered aggregation tree as the tree is less linear; more randomness in good for the aggregation tree algorithms. The k-ordered aggregation tree is slightly more affected by the number of long-lived tuples, as opposed to the aggregation tree, largely because the k-ordered tree has fewer tuples, uses less time, and thus is more affected by more tuples in the tree. Recall that k-ordered relations allow us to garbage collect the node added to the tree from a tuple's start time (within a certain number of nodes depending on k). There may also be a node added to the tree for the tuple's end time. If the end time is far away from the start time (i.e., the length of the tuple in time is large), then many tuples will have to be processed in the list before we move past the end time induced tree node. If we only have shorter-lived tuples, then the end time induced tree node will be closer to the start time, and we will be able to garbage collect that node sooner. So, the more longer lived tuples, the greater the number of nodes will be created that will be garbage collected later, as opposed to earlier.

Parodoxically, the aggregation tree's performance improves in the presence of many long-lived tuples. This is because of the size of the relation and the distribution of the length of long-lived tuples. In Figure 7, the tuples are mostly short-lived. Thus few of the tuple insertions into the aggregation tree avoid the construction of a linear list. This linear aggregation tree strictly grows down the right hand side of tree. When many long-lived tuples are present (as in Figure 8), the insertions for the end of the tuple (80% of the tuples are long lived here) result in a less linear right hand side of the tree. The tree is more "bushy". As the algorithm has inserted many tuples into the right hand side of the tree ahead of time, this side of the tree is not linear. Thus we see a performance improvement.

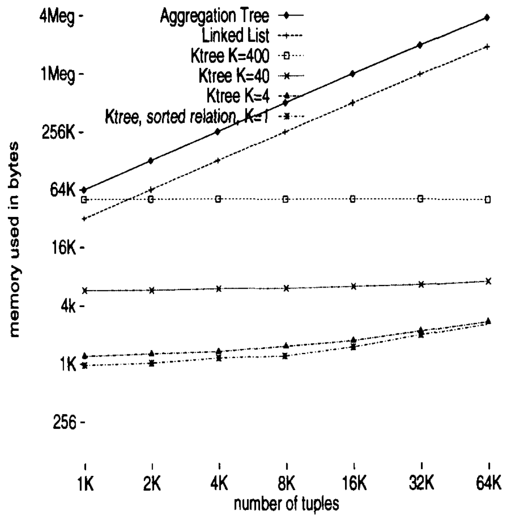


Figure 9: Memory Comparison with No Long-lived Tuples

6.2 Main Memory Comparison

The amount of memory used for a "node" differed between the algorithms. Both aggregation tree algorithms used 16 bytes per node as we implemented the more efficient, single timestamp per node variation: two child pointers, an aggregate-value, and a timestamp split value. The linked list algorithm used 16 bytes per node as it stored two timestamps.

As before, the space requirements of the algorithms vary with percentage of long-lived tuples. Figure 9 compares them under the (unrealistic) assumption of no long-lived tuples. The memory required by the linked list algorithms was basically constant over different k and k-ordered percentage values, so we only present a single result for this algorithm. The memory requirements for the k-ordered aggregation tree when computed over a totally ordered relation (in the legend, *Ktree, sorted relation, K=1*), were barely reduced from the k = 4 tests. The basic aggregation tree requires the most main memory.

For relations with long-lived tuples, the results are much worse for the k-ordered tree algorithms; the memory requirements for the linked list and aggregation tree algorithms are totally unaffected by presence of such tuples.

The memory used by the k-ordered tree algorithm varied for all three factors, but the most important factor was the value of k, closely followed by the percentage of long-lived tuples. The k-ordered-percentage proved to be relatively unimportant in memory usage. This is expected because the ordering of the tuples affects the shape of the tree (and thus the evaluation time), but not the actual number of nodes in the tree (which determines the space usage). The relatively large space requirements for small relations for the k-ordered tree occurs because the (fixed) value of k is such a large percentage of the number of tuples; hence, there is less opportunity for garbage collection.

6.3 Query Optimizer Strategies

The optimizer can exploit information on the sortedness of the underlying relation. If the relation is not sorted, regardless of the number of long-lived tuples, then the aggregation tree algorithm will perform well when compared to the linked list and k-ordered aggregation tree, depending on the tradeoff between the cost of increased memory requirements and the cost of disk access. If memory is cheaper than disk I/O, then the aggregation tree is the best approach. On the other hand, if the relation is sorted, or if the disk access time necessary to sort the relation is less costly than the memory the aggregation tree requires, then the k-ordered aggregation tree is the best approach. If the relation is declared by the data base administrator to be retroactively bounded, then the k-ordered aggregation tree would be the algorithm of choice, as no sorting is required.

The performance of the linked list algorithm was almost unaltered between the various alternatives, depending only on the relation size. Although its performance was poor in comparison to the other algorithms here, it is important to note that if there were very few constant intervals in the results (i.e., we were only interested in the results for a single year and instants represented days), then the linked list algorithm would have quite adequate performance. The number of unique timestamps in the relation has a a similar effect on performance. The tests described in this paper have randomly generated start times, which leads to many unique tuple start times. If there were many fewer unique timestamps, which might be the case if the granularity was very coarse, or if most records were written in a short period of time (e.g., a student-records database with grades all written on the last day of the semester), then less memory would be required to store the "state" for each of the algorithms. This last case would especially improve the memory requirement of the aggregation tree and the linked list algorithms.

7 Summary and Future Work

This paper introduced several new algorithms for computing temporal aggregates which are much more efficient than the linked list algorithm. It also defined two new metrics to quantify temporal relations, k-orderedness and k-ordered percentage.

We introduced the linked list algorithm, an improvement over the only previously implemented temporal aggregation algorithm, which maintains buckets for the results in memory. For relations with only a small number of constant interval results, the linked list algorithm is expected to be the most efficient in time and space. The linked list algorithm was unaffected by long-lived tuples. However, the linked list algorithm was still slower than either of the other new algorithms under all tested conditions.

For unordered relations, we introduced the aggregation tree, which builds a binary tree of the constant intervals, and showed that this algorithm is the most efficient in time, dependent in part on the number of long-lived tuples. The space usage of the aggregation tree is generally greater than the linked list algorithm, due mostly to the fact that each unique timestamp adds two nodes to the aggregation tree and only one in the case of the linked list algorithm.

For k-ordered relations, we introduced the k-ordered aggregation tree, a variation of the aggregation tree with garbage collection of tree nodes. This algorithm was generally the most efficient for k-ordered relations with any long-lived tuples. We tested this algorithm with different values of k and k-ordered-percentages and found that the k-ordered aggregation tree worked best for small values of k and for larger k-ordered-percentages. The linked list algorithm was unaffected by different values of k and k-ordered percentages.

To summarize, we have presented techniques for computing temporal aggregates for unordered, fully sorted, and almost ordered temporal relations, and empirically shown under what conditions each of the algorithms is best. The simplest strategy is to sort the relation then use the k-ordered aggregation tree with $k = 1$. This gives very efficient run-time performance across a range of long-lived tuple percentages, with minimal memory usage. When the relation is not ordered, but is retroactively bounded, then the k-ordered aggregation tree is directly applicable without sorting.

There are several further areas of research to explore in temporal aggregation. One alternative to examine is a balanced aggregation tree, which should be especially efficient in the case of a k-ordered relation.

Another possibility for future research concerns the aggregation tree. If the relation *might* be sorted, then the best choice would be the aggregation tree algorithm, with the relation's pages randomized when they are read to avoid linearizing the aggregation tree. This randomization could be performed on each group of pages read into memory, and therefore would not affect the I/O time.

Another aspect to investigate is temporal grouping by span. If the number of spans is much smaller than the number of constant intervals, then fewer "buckets" need to be maintained as there will be many fewer constant interval results. The performance of the slower algorithm tested here (the linked list) would be expected to improve.

We did not consider duplicate elimination. This will probably not affect the linked list algorithm very much, but is another matter entirely for the tree algorithms. Our choices depend on the number of tuples in each interval. Probably the best single approach for this problem involves removing the duplicates before the relation is processed, perhaps by sorting.

Finally, we want to explore limited main memory implementations of these algorithms. The performance of the aggregation tree appears to be a promising alternative for true randomly ordered relations, but the memory requirements are excessive.

The techniques described here may also be applied to spatial and spatiotemporal databases to compute aggregates and associate them with intervals in space and time.

Acknowledgements

Mike Soo, Suchen Hsu and Curtis Dyreson provided helpful comments. We thank Sampath Kannan for his suggestions and discussions of metrics for quantifying the sortedness of the underlying relation, and Andrey Yeatts for his suggestions regarding the implementation of the garbage collection algorithm.

This work was supported in part by NSF grants IRI-8902707 and IRI-9302244, and also by an AT&T Foundation Grant.

Bibliography

[Bitton et al 1983] Bitton, D., H. Boral, D. DeWitt and W.K. Wilkinson. "Parallel Algorithms for the Execution of Relational Database Operations." *ACM Transactions on Database Systems*, 8, No. 3, Sep. 1983, pp. 324–353.

[Ceri & Gottlob 1985] Ceri, S. and G. Gottlob. "Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries." *IEEE Transactions on Software Engineering*, SE-11, No. 4, Apr. 1985, pp. 324–345.

[Dyreson & Snodgrass 1993] Dyreson, C. E. and R. T. Snodgrass. "Timestamp Semantics and Representation." *Information Systems*, 18, No. 3 (1993), pp. 143–166.

[Dyreson & Snodgrass 1994] Dyreson, C. E. and R. T. Snodgrass. "Temporal Granularity and Indeterminacy: Two Sides of the Same Coin." Technical Report TR 94-06. Computer Science Department, University of Arizona. Feb. 1994.

[Epstein 1979] Epstein, R. "Techniques for Processing of Aggregates in Relational Database Systems." UCB/ERL M7918. Computer Science Department, University of California at Berkeley. Feb. 1979.

[Freytag & Goodman 1986] Freytag, J.C. and N. Goodman. "Translating Aggregate Queries into Iterative Programs," in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayaski. Kyoto, Japan: Aug. 1986, pp. 138–146.

[Gray 1991] Gray, Jim (ed.) "The Benchmark Handbook for Database and Transaction Processing Systems." Morgan Kaufmann, 1991.

[Jensen & Snodgrass 1994] Jensen, C. S. and R. Snodgrass. "Temporal Specialization and Generalization." *IEEE Transactions on Knowledge and Data Engineering*, (1994).

[Jensen et al. 1994] Jensen, C. S., J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes and S. Jajodia [eds]. "A Glossary of Temporal Database Concepts." *ACM SIGMOD Record*, 23, No. 1, Mar. 1994, pp. 52–64.

[Kiessling 1985] Kiessling, W. "On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates," in *Proceedings of the Conference on Very Large Databases*. Ed. A. Pirotte and Y. Vassiliou. Stockholm, Sweden: Aug. 1985, pp. 241–250.

[Kim 1982] Kim, W. "On Optimizing an SQL-like Nested Query." *ACM Transactions on Database Systems*, 7, No. 3, sept 1982, pp. 443–469.

[Kline 1993] Kline, N. "An Update of the Temporal Database Bibliography." *ACM SIGMOD Record*, 22, No. 4, Dec. 1993, pp. 66–80.

[Kline et al. 1994] Kline, N., R. T. Snodgrass, and T.Y. Leung. "Aggregates for TSQL2." Commentary. TSQL2 Design Committee. Sep. 1994.

[Preparata & Shamos 1985] Preparata, F. P. and M. I. Shamos. "Computational Geometry, An Introduction." Springer-Verlag, 1985.

[Snodgrass et al. 1993] Snodgrass, R., S. Gomez and E. McKenzie. "Aggregates in the Temporal Query Language TQuel." *IEEE Transactions on Knowledge and Data Engineering*, 5, Oct. 1993, pp. 826–842.

[Snodgrass et al. 1994] Snodgrass, R.T., I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Kafer, N. Kline, K. Kulkanri, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. "TSQL2 Language Specification." *ACM SIGMOD Record*, 23, No. 1, Mar. 1994, pp. 65–86.

[Tansel et al. 1993] Tansel, A., J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (eds.). "Temporal Databases: Theory, Design, and Implementation." Database Systems and Applications Series. Redwood City, CA: Benjamin/Cummings, 1993.

[Tuma 1992] Tuma, P. A. "Implementing Historical Aggregates in TempIS." Master's Thesis, Wayne State University, Nov. 1992.

[vonBultzingsloewen 1987] vonBultzingsloewen, G. "Translating and Optimizing SQL Queries Having Aggregates," in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 235–243.