Computer Sciences Department

Aggregate Sharing in Stream Databases

LidanWang
Jeffrey Freschl

Technical Report #1536

October 2005

UNIVERSITY OF WISCONSIN MADISON

Aggregate Sharing in Stream Databases

Lidan Wang, Jeffrey Freschl
{lidan, jfreschl}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin, Madison
May 13, 2005

Abstract

We consider the problem of handling aggregate computations in a scalable fashion in stream databases. The queries of interest are sliding-window aggregate queries over a single data stream. In the naïve approach for this problem, each query is individually processed one at a time, such that no sharing occurs among aggregates in terms of storage and answer computations. We present two new algorithms, *TimeGram* and *OrderedSharing*, both of which are designed with sharing among aggregates in mind. Our performance results show that both drastically improve the current performance of STREAM, a stream database system for which we have implemented and evaluated our two algorithms.

1. Introduction

Traditional DBMS is not designed for continuous arrival of data tuples and continuous queries. A traditional DBMS only supports data that is in the form of persistent relations, and therefore it will not work for applications in financial, network monitoring, or sensor networks. In contrast to the traditional DBMS, a stream database system can handle both continuous data and continuous queries, a feature that has drawn attention from many researchers. Much research has been devoted to how to process data tuples and answer queries in a scalable fashion when there are 1000s to 10,000s queries in the stream database system. Techniques for sharing selection predicates are fairly well understood, but sharing aggregate computations in a scalable manner is largely unstudied.

In this paper we present two new algorithms, *TimeGram* and *OrderedSharing*. They aim to improve scalability and response time in STREAM. The update, look-up, space, and response time have been greatly reduced due to opportunities for sharing in sliding-window aggregates, both in terms of storage and answer computations. By exploiting them we obtain drastic performance improvement over naïve, a non-sharing approach used in STREAM. The remainder of the paper is organized as follows; in section 3 we introduce TimeGram, and in section 6 we introduce OrderedSharing.

2. An Overview of the Naïve Approach in STREAM

The current naïve algorithm in STREAM processes each query individually. Each query is assigned a storage space that is used to store data tuples. Also, the answer for a query is computed independently of other queries, even though some queries may have the same aggregate attribute. This naïve approach also ignores the fact that data tuples are common to all sliding-window queries. As a result, a large space overhead and computation overhead occur in the system.

Furthermore, in addition to updating data structures with new tuples, the naïve approach also evaluates queries upon *every* new arrival of data tuples. This is an undesirable and unnecessary property since we only want to lookup answers when the users request them.

The pseudo code for the current naïve approach is given below.

```
Setup stage:
For each sliding-window query q_i in the system
   Assign an input queue and an output queue to q_i
End

Data processing and evaluation stage:
Insert (tuple t)
   For each window in the system
      Store tuple t into the window's input queue
      Remove the oldest tuple from the input queue if necessary
      Update/compute answer, and stream out the answer to output queue
   End
End
```

Figure 1 Naïve approach

3. The TimeGram Algorithm

We present the TimeGram algorithm and illustrate it through an example. But first, we will give some background information on STREAM.

3.1 Data Tuples and Timestamps in STREAM

Each data tuple in STREAM has a timestamp in addition to attribute values. If tuple a arrives before tuple b, then a's timestamp is strictly less than b's timestamp, i.e. ts_a < ts_b. A stream of data is just an ordered sequence of data tuples, <ts_a, a>, <ts_b, b>, ... <ts_z, z>. Here a, b, ...z represent the tuples themselves, and ts_i, i=a, b, ...z, represents the timestamps.

Users can issue aggregate queries such as Select Max From S(A) [Rows 6]. It says "find the maximum value on attribute A from the last 6 tuples in stream S". Here the window size for this sliding-window query is 6. Users can also issue other types of aggregate queries such as Min, Count, Sum, Avg.

3.2 The TimeGram Algorithm

3.2.1 Key Points

1) It builds a "timegram" on the timestamps of the tuples in the largest window. The timestamps are put into appropriate buckets based on their tuple values on the aggregate attribute.
2) We exploit sharing since the single timegram data structure is shared by all queries for both tuple processing and answer computations.

3) We only need to build the timegram on the largest window, because this timegram will contain all the information for answering queries on any smaller windows in addition to the largest window. This is due to the fact that the data of any smaller window is included in the data of the largest window.

4) The answers are approximated but can be made more precise by adjusting bucket width. For a given query the answer is returned as [l, r], the boundary values of a bucket in which the exact answer resides.

The timestamp and bucket information plays a very important role in this algorithm, in terms of how to answer queries. We give the algorithm and an example below.

3.2.2 How to Build a TimeGram and How to Process Tuples

The steps for building a timegram and processing tuples are as follows:

a) Given a collection of aggregate queries on some common attribute A in stream data S, each query is a sliding-window query Qi and therefore has a window size Wi.

b) Find the value range for A (aggregate attribute) in the largest window. For example, let it be [L, R], where L and R are the minimum and maximum values, respectively, for attribute A in the largest window.

c) Uniformly divide [L, R] into K buckets, B1, B2, ...Bk, each with width [R-L]/K.

d) When a new tuple arrives, its timestamp is put into an appropriate bucket based on the tuple's aggregate attribute value. For example, if tuple 3 arrives and its aggregate attribute value is -1, and if a bucket B2 has range [-2, 4], then the timestamp of tuple 3 will be stored in bucket B2.

e) If with the addition of the new tuple's timestamp the total number of timestamps held in the timegram data structure is greater than the max window size, remove the oldest timestamp.

f) At any given time, buckets B1, ... Bk hold the timestamps of tuples in the largest window.

3.2.3 How to Lookup Answers

There are five cases for answer lookups. Let's assume the window is W in a query we want an answer for.

a) **Max:** start from the last bucket (which holds the timestamps of tuples with the largest values on the aggregate attribute), check if any timestamp in this bucket belongs to W. If it is, return this bucket's boundary values [l, r] as the answer. If no timestamp in

this bucket belongs to window W, move down to the next bucket whose boundary values are smaller, repeat, until we find the first appearance of the timestamp that belongs to this window W.

b) **Min:** similar to Max. Instead of starting from the last bucket, we start from the first bucket, and the rest of the details are the same.

c) **Count:** if the query is on the largest window, just return the total number of timestamps stored. Otherwise, we should check each bucket to count the number of timestamps that belong to this W (we can use an efficient data structure to do this).

d) **Sum:** use bucket boundary values and Count results. Let Bucket 1=[L1, R1], and Bucket 2=[L2, R2], count_i=number of timestamps of W in bucket i, return the answer as Sum=[a, b] where a=L1*count_1+L2*count_2, and b=R1*count_1+R2*count_2.

e) **Avg:** similar to Sum, just divide the answer for Sum by the total number of timestamps in the timegram data structure.

3.2.4 An Example

Assume we have the following stream data source. The format of each tuple is <timestamp ts, attribute A, attribute B>.

Stream S:
T1=<1, 3, atc>
T2=<2, -10, agg>
T3=<3, -9, gtt>
T4=<4, 5, acc>
T5=<5, 2, ttg>
T6=<6, 7, gga>
T7=<7, -3, aat>

Assume there is a collection of standing aggregate queries as below:

Query 1: Select Sum (A) From Stream S on window size 4
Query 2: Select Max (A) From Stream S on window size 3
Query 3: Select Avg (A) From Stream S on window size 2

Build a timegram and lookup for answers:

Observe the aggregate attribute value range is [-10, 10] and max window size is 4.

Define 5 buckets (note the number of buckets here is user-defined):
Bucket 1: [-10, -6]
Bucket 2: [-6, -2]

Bucket 3: [-2, 4]
Bucket 4: [4, 6]
Bucket 5: [6, 10]

Process Data (only store timestamps for tuples in the largest window):

The tuples arrive in a continuous manner.

T1 arrives: T1's A attribute = 3 ➔ put the timestamp 1 in Bucket 3
T2 arrives: T2's A attribute = -10 ➔ put the timestamp 2 in Bucket 1
T3 arrives: T3's A attribute = -9 ➔ put the timestamp 3 in Bucket 1
T4 arrives: T4's A attribute = 5 ➔ put the timestamp 4 in Bucket 4
T5 arrives: T5's A attribute = 2 ➔ put the timestamp 5 in Bucket 4
Total number of timestamps held by buckets > max window size 4
Remove timestamp for T1
T6 arrives: T6's A attribute = 7 ➔ put the timestamp 6 in Bucket 5
Total number of timestamps held by buckets > max window size 4
Remove timestamp for T2
T7 arrives: T7's A attribute = -3 ➔ put the timestamp 7 in Bucket 2
Total number of timestamps held by buckets > max window size 4
Remove timestamp for T3

In the end each bucket's content is as follows:

Contents:					
		T7		T5 T4	T6
Buckets:	<-10, -6>	<-6, -2>	<-2, 4>	<4, 6>	<6, 10>

Figure 2 Resulting bucket content

Observations:
1. Some buckets may be empty and thus when we lookup answers we can just skip them.
2. Constant storage: the space is bounded by the number of tuples in the largest window

Lookup Answers:
If we want the answer for Query 2: Select Max (A) From Stream S on window size 3, we check if <6, 10> contains a timestamp for window size 3, which it does (since window size 3 has timestamps T7, T6, and T5), so we just return <6, 10> as an answer for this query (the exact answer is 7 which lies within this interval).

Important Note:
Note that tuple processing and answer lookups can be an interleaved process; i.e., we can look for answers in the middle of processing tuples, although for simplicity we did not illustrate this in the example.

3.2.5 Time and Space Complexity

The TimeGram algorithm is a highly scalable algorithm with O(1) update cost and O(1) lookup cost (if using efficient data structures for buckets), and a space cost that is bounded only by the max window size in the standing queries, O(MaxWin).

4 Key Advantages of TimeGram

1. Highly scalable due to one common timegram data structure that is shared by all queries for tuple processing and answer computations.
2. Cheap lookup O(1) and update O(1)
3. Reduced space overhead; unlike the Naïve approach (separate storage for each query in DB), we only use one common shared storage, the timegram data structure (i.e. a list of buckets).
4. Adaptive to user needs for answer precision and answer frequency; we can change bucket width to get more/less answer precision, and can also adjust the update_lookup ratio to achieve different answer lookup frequencies.

5 Experiments and Analysis:

In this section we will look into how TimeGram performs in comparison to the Naïve algorithm. We think that there are three key factors that govern the response time for the system, and they are:

1) Update_lookup ratio (how often a user wants query answers versus how often new data tuples are processed).
2) The total number of queries in the system
3) Skewed data distribution

First we will use 1000 randomly generated data tuples to illustrate point 1) and point 2), and then we will repeat the same experiments by using 1000 skewed data tuples. Also, when evaluating point (2) we were restricted to a maximum of 8 queries due to limitations in the system; ideally, we would have liked to register as many as 1000 queries.

5.1 Response Time vs. Update_Lookup Ratio

In both algorithms new data tuples are processed right away as they continuously arrive. However, the lookup rate is different between TimeGram and Naïve. The Naïve algorithm re-evaluates aggregate answers whenever a new tuples arrives, because both the current aggregate answers and the new tuple have to be used for computing new aggregate answers. This scheme is inefficient for two reasons:

1) First, it is unnecessary to give query answers when users did not ask for them, and it wastes system resources in doing so.

2) Second, it binds its evaluation scheme with one particular update_lookup ratio, 1:1. This is not a desirable property because the update_lookup ratio is really a system parameter, and therefore the algorithm should not be developed based upon one particular value of this parameter.

TimeGram resolves both of these two issues. The update_lookup ratio now is a parameter of the TimeGram algorithm and the lookup now becomes on-demand.

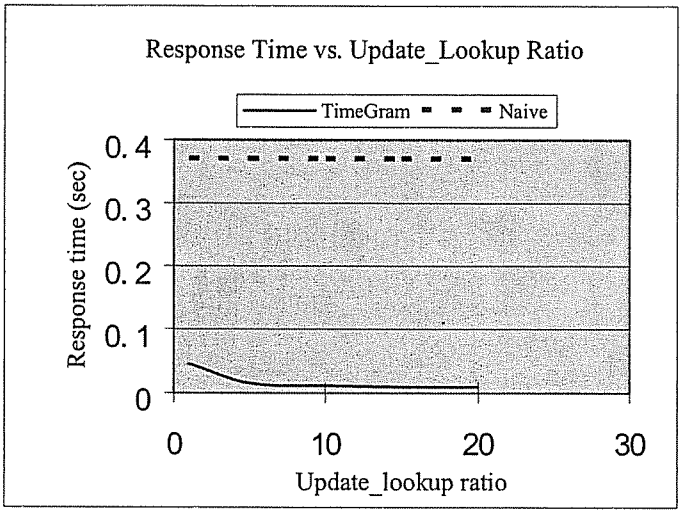

Figure 3 Response time vs. update_lookup ratio

As we see from Figure 3, the response time for Naïve is constant because it always re-evaluates on each new tuple arrival. For TimeGram, the response time gradually decreases as lookup becomes more sporadic, and the difference between the two algorithms also becomes greater as a result of more infrequent lookups. Also, TimeGram is much faster and uses less system resources (due to the sharing among aggregates) than Naive.

5.2 The Total Number of Queries in the System

TimeGram handles aggregate sharing in a scalable fashion. All aggregate queries (with the same aggregate attribute) share one common data structure for tuple processing and answer computations. The data structure, i.e. a list of buckets as we saw earlier, holds all the necessary information for answering all queries, and its space is only bounded by the size of the largest window, not by the total amount of data tuples. TimeGram is highly scalable with a large amount of queries.

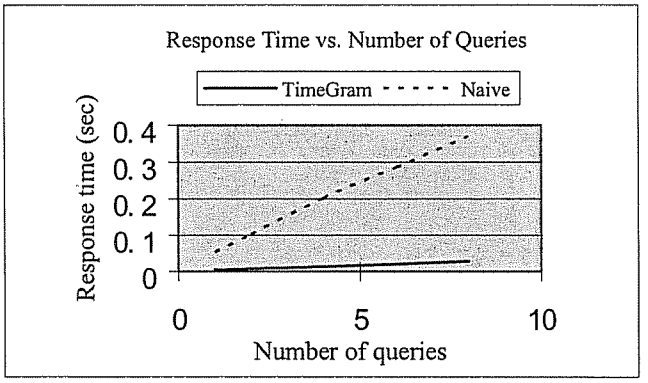

Figure 4 Response time vs. number of queries [1]

As we see from figure 4, the response time increases as the number of queries grow. The rate of increase, however, is much larger for Naïve than for TimeGram, as evidenced by the sharp slope exhibited by Naïve and a relatively flatter slope exhibited by TimeGram. In other words, this has confirmed our earlier belief that TimeGram scales well as the amount of queries grows due to the way it handles aggregate sharing.

5.3 Skewed Data Distribution

We are also interested in learning how skewed data would impact the results we have seen. The data tuples are now generated using a Gaussian probability function that produces 1000 skewed data tuples, most of them are between [-20,20]. We then repeat the previous two experiments.

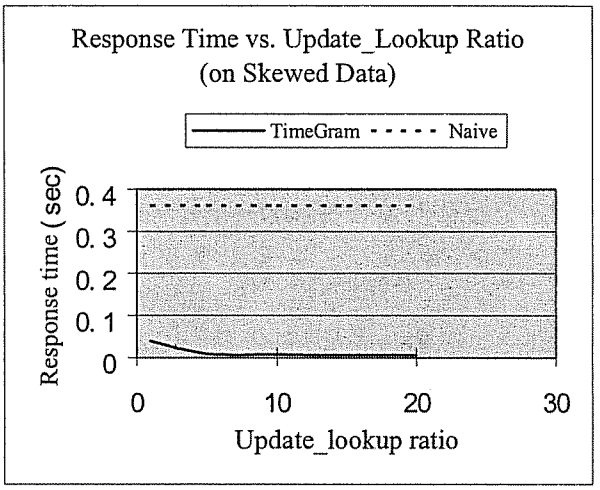

Figure 5 Response time vs. update_lookup ratio on skewed data

[1] With the current STREAM it cannot support more than 8 user queries. But nonetheless, our claims on increased scalability can still be illustrated regardless of this issue.

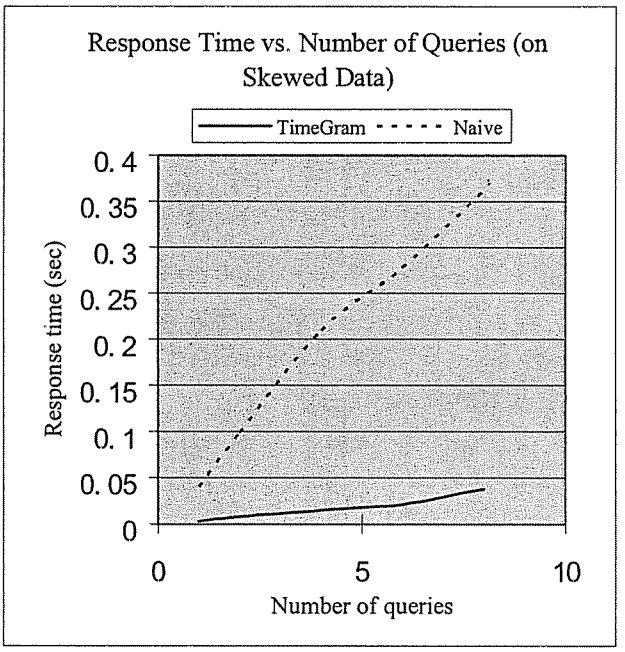

Figure 6 Response time vs. number of queries on skewed data

As we see from Figures 5 and 6, the relative performance between TimeGram and Naïve is about the same as before. TimeGram has a better response time, which is inversely proportional to the lookup frequency. It also scales well with a large number of queries. This robustness is due to the implementation details of TimeGram. Skewed data may result in unbalanced distribution of data tuples such as some buckets may be empty or have fewer tuples than others. However, this is not a problem because each bucket has a size variable, and if the size is zero for some bucket the algorithm immediately skips that bucket. More importantly, the amount of timestamps inspected during the course of execution on skewed data is really the same as before on randomly generated data. As a consequence, the placement of data does not have a noticeable impact.

5.4 Comments on TimeGram's Bucket Size

Since for a given query, TimeGram returns an approximate answer in terms of the two boundary values of a particular bucket in which the exact query answer resides, we can make the answer more precise or more vague by reducing or expanding the bucket size, respectively. Thus, TimeGram can adapt to user needs on answer precision, i.e. in certain applications users may want a less precise answer in the interest of a low time cost, while in other applications users may want more precision and do not care about costs. TimeGram can cater to both situations.

6 The OrderedSharing Algorithm

In this section, we introduce OrderedSharing, a highly scalable algorithm that relies on two key properties of a set of windows to maximize the amount of sharing between

aggregate queries. For clarity, we assume that the aggregate function is max, although we do show how to support min. Unlike in TimeGram, subtractive aggregates are not supported.

6.1 Preliminaries

In this section we discuss the state of each window, and also define what it means for a window to be valid.

For each window w, we have

1. An answer for this window (denoted w.answer)
2. A timestamp ts for when we calculated w.answer (denoted w.ts). Initially, the ts is –infinity meaning that we cannot use the answer.
3. The size of this window (denoted w.size); that is, how many tuples fit into this window.

Definitions:

Valid: a window w is valid at time currentTime iff there exists a window w' such that w'.size >= w.size and

$$\text{currentTime} - w'.ts < w.size$$

This means that we either have an answer w.answer for w that is within its window, or we have an answer for a larger window w' that is also contained within the window of w.

Invalid: a window that is not valid.

6.2 Key Ideas

Given a set of n windows $W = \{W\_i\}$, we can order the windows by size in increasing order; i.e.,

$$W = \{W\_1, W\_2, \ldots, W\_n\} \text{ s.t. for any } i < j,\ W\_i.size < W\_j.size \quad (1)$$

Observe that because increasing the size of a window can only include more tuples, we also have that the answer can only increase in value; i.e.,

$$\text{For any } i < j,\ W\_i.size < W\_j.size \rightarrow W\_i.answer < W\_j.answer \quad (2)$$

6.2.1 Insert

Because of the above properties (1) and (2), if a new arrival tuple is the max of a window w, then it is also a max of all smaller windows. Specifically, on an insert of value A, we first find the largest valid window w such that w.answer <= A. We then validate window w and all smaller windows w' (i.e., w'.size <= w.size) by updating w.answer and w.ts to

A and the currentTime respectively. So that we do not have to update the answer and timestamp for all smaller windows with the same value, we set their timestamps to – INFINITY (this takes 1 operation, described below in our implementation). Note that if we did not use *the* largest window, then we would have a valid window with a wrong answer which is incorrect.

In summary, given a set of windows, some of which we have answers for (i.e., the valid windows), and given a new arrival value, does this value change the answer of any of the current valid windows? All that matters is that all valid windows have the correct answer. All invalid windows will be handled accordingly on a lookup.

6.2.2 Lookup

Initially all windows are invalid and will remain invalid until a lookup is performed on some window w. Remember that the only way we can implicitly validate an invalid window is if it can use the answer of some larger valid window. To explicitly validate an invalid window, we must re-calculate its answer. Subsequently, the lookup will then find that w is invalid and therefore it must calculate the answer for w by iterating over all tuples in w. A nice consequence of this is that while calculating the answer for w, we are also calculating the answer for all smaller windows. Thus, we not only validate w, we also validate all smaller windows! Here is a summary of what happens when a lookup is performed:

Lookup(window w)

1. Is w valid?
2. If so, return the answer for w.
3. otherwise, re-calculate the answer for w and all smaller windows.

6.3 Implementation

(1) On insert(value A), how do we efficiently find the largest valid window w such that w.answer <= A without having to iterate over each window? In other words, given an ordered set of valid and invalid windows, how do we find a window with a specific answer?

(2) Similarly, on a lookup(window w), how do we find a window w' that has an answer valid for w? In other words, how do we find a window with a specific size?

6.3.1 Main Data Structure – Binary Search Tree

The data structure we use is a *single* Binary Search Tree sorted on answers. To answer question (1) on an insert of value A, we do a binary search for A to find a current valid answer for some window w, such that w.answer <= A. A nice property of this tree is that because it is sorted on answers where each answer has a corresponding window, the tree is also sorted on the window size. Remember properties (1) and (2) give that the

windows are sorted on size if and only if their answers are also sorted. In other words, if value w1.answer1 < w2.answer2, then w1.size < w2.size.

6.3.2 Insert and Lookup on the Tree

In this section, we give a summary of how to insert and how to do a lookup within the binary tree.

6.3.2.1 Insert

Insert(Tree t, newValue, currentTime)

1. Find the largest valid answer w.A in t such that w.A <= newValue, where A is the current answer for window w. (This is a simple binary search for newValue in tree t)
2. If value w.A does exist
    a. w.answer ← newValue (since w.A now has a new answer)
    b. w.ts ← currentTime (We changed w's answer at time currentTime)
    c. Remove all smaller windows r from the tree (i.e., r.size < w.size), because their answer is now stored in w.answer.
    d. Fix Tree if necessary. In some cases, updating w.answer violates the binary search tree property. This occurs in 2 cases: (1) The right child of w is invalid, and thus can have a smaller answer after the update; (2) The parent of w is invalid, and thus can have a smaller value.
3. otherwise, all valid answers are greater than newValue, and thus there are no answers to update.

Example 1) Assume all nodes are valid in the following tree:

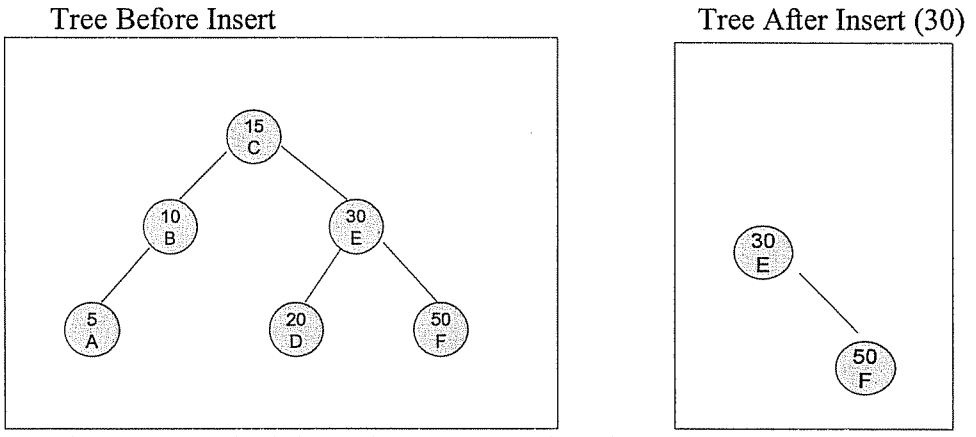

Figure 7. On the left, we have answers for windows A, B, C, D, E, and F. Notice that there answers are also sorted! After an Insert of value 30, we get the tree on the right.

What happens when we insert value 30 into the tree? Step (1) of Insert gives us window E. Then for step (2) we update the answer and timestamp of window E, and for step(3) we remove all smaller windows giving us the tree on the right in Figure 7, because not only is 30 an answer for E, it is an answer for all smaller windows.

For the tree before insert, what happens if E is invalid when inserting value 35? Since window D has the largest valid answer smaller than 35, we update D to have 35 as its answer, and then remove A, B, and C (i.e., all smaller windows). However, in this case, increasing the answer of node D violates the binary search tree property, because E has a smaller answer than its left child. Thus, we must fix the tree by removing node E, and replacing it with node F. We would then replace F with its right child if the tree property was still violated; however, this only occurs if F is also invalid.

6.3.2.2 Lookup

// Find the answer for window w

Lookup(Tree t, window w)

1. if w is valid
    a. return answer for w.
2. otherwise,
    a. remove all windows smaller than w, because we are going to re-calculate the answer for all smaller windows.
    b. Re-calculate the answer for w and all smaller windows. In our actual implementation, we used a Red Black Tree to keep the tree balanced.
    c. For each window w' we validate, we insert <currentTime, w'.answer, w'.size> into the tree with its new answer.

Example 2)

Assume all nodes are valid in the following tree t:

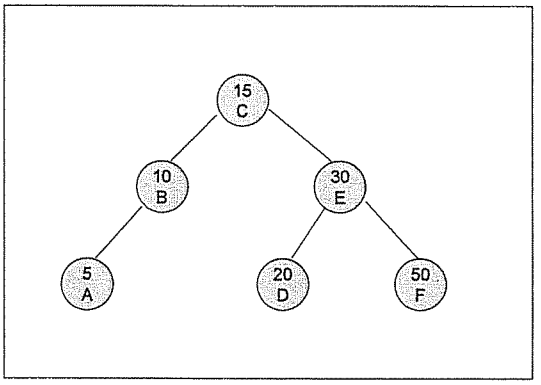

Figure 8: Here we have answers 5, 10, 15, 20, 30, and 50 for windows A, B, C, D, E, and F respectively.

What happens when we perform a lookup on window E in Figure 8? Doing a binary search in t, we find that not only is E valid, but that it is also in the tree; thus, we return 30 as an answer. What if we did a lookup on window E' where E.size < E'.size < F.size? In this case, F is the smallest window larger than E'. We then see if F.answer is valid for us. If it is, then we return 50, otherwise, the value 50 is outside the window of E' and we re-calculate the answer of E'. Of course, for every window we validate, we must first remove it from the tree and then re-insert the window with its new answer.

6.3.2.3 A Single Binary Tree for Min

Using a single binary tree to store both the window size and the answer works for max, because with increasing answers, we also have increasing window sizes. However, does this work for min, where with increasing answers, we have decreasing window sizes? Using a single tree still works, because the only difference is how to search for a window within the tree. Basically, on a lookup(window w), if w > currentTreeNode (i.e., the window size of the current node in the binary search), then go to the left child which will lead us to a smaller answer, but larger window. The window size must be larger, because adding more tuples (i.e., increasing the window size) can only give us a better answer (i.e., a smaller min value).

6.4 Time and Space Complexity

The time to do an Insert is O(log(|w|)), where |w| is the number of registered windows. The reason is that the main operation in Insert is to do a binary search for the new inserted value. Similarly, the time to perform a Lookup is O(log(|w|)), because we are doing a binary search on the looked up window. Finally, the space complexity is the space to store the binary tree, which is O(|w|) in the worst case, although in the field the space may be much smaller depending on the window size distribution.

6.5 Experiments and Analysis

In this section, we will look into how OrderedSharing performs in comparison to the Naïve algorithm. We repeat the experiments in section 5, except that we replace TimeGram with OrderedSharing.

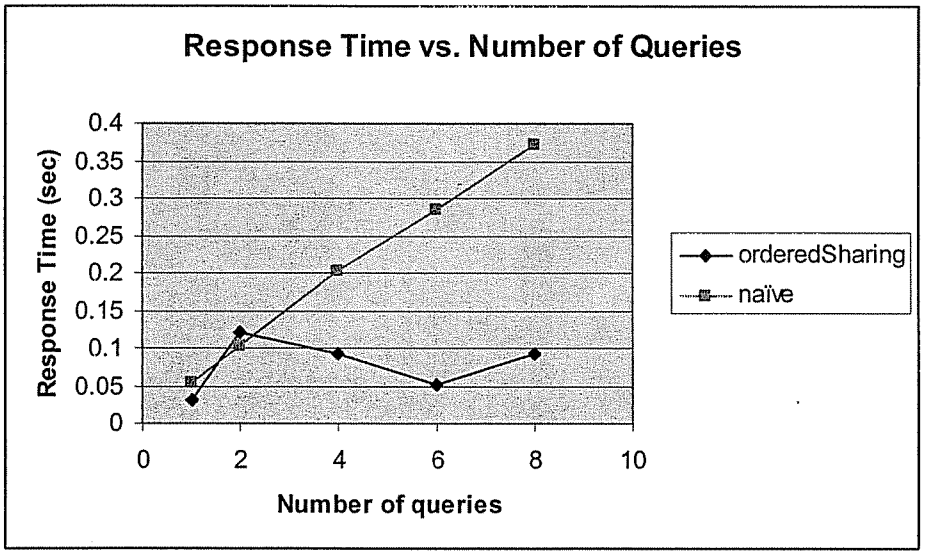

Figure 9: Response Time vs. Number of Queries.

In Figure 9, we see that OrderedSharing performed quite well in comparison to naïve. After 2 queries, the response time for orderedSharing remains relatively constant, while the response time for naïve steadily increases. The reason is that for OrderedSharing, when a single window w is validated, most of the other windows could use w's answer for a large number of lookups until eventually w's answer is no longer valid. Then in the next few lookups, a majority of the windows are once again valid, thus requiring a small lookup time for subsequent lookups.

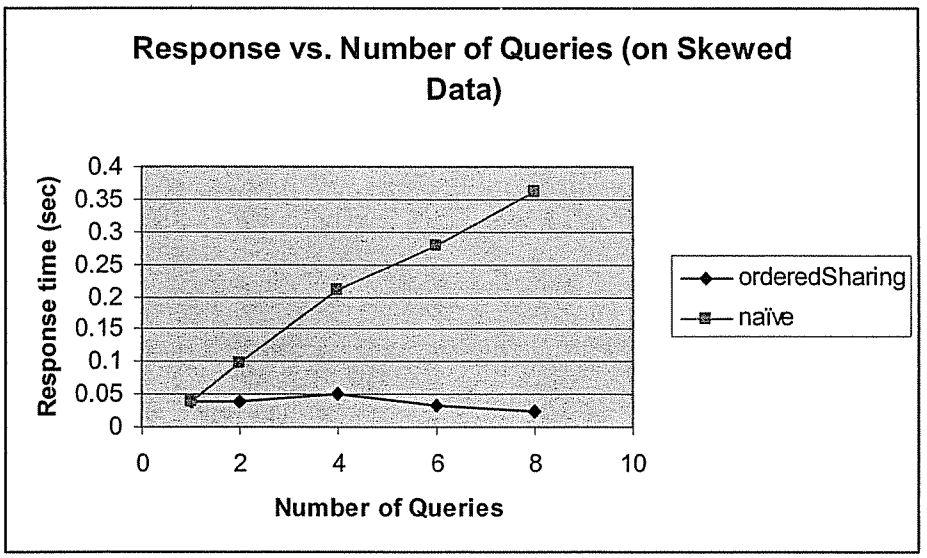

Figure 10: Response Time vs. the number of queries on skewed data.

In Figure 10, we see that OrderedSharing performed slightly better than before with non- skewed data. The reason for this slight improvement is that because the new

arriving tuples are skewed, once the maximum for a window is established, it remains the maximum for a longer time making the tree and its contained answers more stable.

7 Conclusions

We have introduced the TimeGram algorithm, which is a scalable algorithm for shared aggregate computations in stream database systems. It builds a timegram data structure which is shared by all queries for both tuple processing and answer computations. The answer is returned as <l, r>, boundary values of a bucket in which the exact answer resides. The TimeGram has a much lower response time than Naïve and it can adapt to user needs on answer frequency and answer precision, in addition to being scalable and having low lookup, update, and space overhead.

We also introduced the OrderedSharing algorithm that stores the answers for only a subset of the registered set of windows. On an Insert, we would have to find which window's answer if any, needed to be updated. This would also imply that all smaller windows had the same answer as the largest window whose answer was updated allowing us to only store the largest window's answer. We were then able to use a *single* Binary Search Tree sorted on the window's answers, to either find a specific window size or answer in log time. Finally, in our performance analysis we were able to show that OrderedSharing performs much better than Naïve with the increase in number of concurrent queries.