On-the-Fly Sharing for Streamed Aggregation *

Sailesh Krishnamurthy
Computer Science Division
Department of EECS
UC Berkeley, CA 94720
sailesh@cs.berkeley.edu

Chung Wu
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
chungwu@google.com

Michael J. Franklin
Computer Science Division
Department of EECS
UC Berkeley, CA 94720
franklin@cs.berkeley.edu

ABSTRACT

Data streaming systems are becoming essential for monitoring applications such as financial analysis and network intrusion detection. These systems often have to process many similar but different queries over common data. Since executing each query separately can lead to significant scalability and performance problems, it is vital to *share* resources by exploiting similarities in the queries. In this paper we present ways to efficiently share streaming aggregate queries with differing periodic windows and *arbitrary* selection predicates. A major contribution is our sharing technique that does not require any up-front multiple query optimization. This is a significant departure from existing techniques that rely on complex static analyses of fixed query workloads. Our approach is particularly vital in streaming systems where queries can join and leave the system at any point. We present a detailed performance study that evaluates our strategies with an implementation and real data. In these experiments, our approach gives us as much as an order of magnitude performance improvement over the state of the art.

1. INTRODUCTION

Data streaming systems are increasingly used as infrastructure for critical monitoring applications such as financial alerts and network intrusion detection. At Berkeley, we have built such a system, TelegraphCQ [6], in the context of the HiFi [12] project that is aimed at managing data from widely dispersed receptors.

These monitoring applications often have many concurrent users asking similar but different queries over a common data stream. For example, a system that monitors stock market trades might have multiple users interested in the total value of trades in a sliding window. While some of these users might care about stocks of a particular sector, or only about high volume trades, others might compute complex user-defined predicates (as suggested in CASPER [9]) on fluctuating quantities like stock price. Similarly, the aggregation window that different users are interested in can vary widely. Money managers in financial institutions who run algorithmic trading systems might want aggregates over 5-10 minute windows reported every 60-90 seconds depending on the specific financial models they use. In contrast, day traders with individual investing strategies might only need these results every 5-10 minutes. Clearly such a system will have to support hundreds of queries, each with different predicates, *and* sliding windows.

Using a naïve approach to execute such queries separately can lead to scalability and performance problems as each additional query adds significant load to the system. When the load becomes too high, the system has to either limit the number of queries, or resort to load shedding [31] and drop tuples. Instead, we exploit similarities in the queries to *share* resources and thus scale better.

Aggregate Queries		Techniques	
Predicates	Windows	Shared	Section
Same	Different	Slices	3
Diffferent	Same	Fragments	4
Different	Different	Shards	5

Table 1: A staged approach to shared aggregation

In our scenario, a stream processor needs to support hundreds of aggregate queries that differ in their predicates *and* windows. In this paper, we attack this problem in stages, by considering in turn query sets with the characteristics shown in Table 1. The table lists the technique we have developed for each case, and the section that explains it. We first focus on queries that differ only in their window specifications. Next we show how to share aggregate queries that have varying selection predicates. Finally, we put these techniques together to solve our problem of sharing aggregate processing for queries with different windows *and* predicates.

Shared processing has been studied for Multiple Query Optimization (MQO) in static databases [29, 28, 18, 10], as well as with streams [7, 23, 21]. These earlier approaches statically analyze a fixed set of queries in order to find an optimal execution strategy. We argue against this "compile-time" approach for two reasons:

1. **Dynamic Environment.** Queries can join and leave a streaming system at any time. A static approach would require expensive recompilation (Aurora [4]) at each such event, consuming precious system resources.
2. **Complexity of Analysis.** It turns out that for the specific problem that we consider, i.e., shared aggregation with varying windows and selections, static analysis is prohibitively expensive. This is because of the high cost of analysis for varying windows (Section 3.3), and because of the complexity and unknowns in isolating common sub-expressions in a set of arbitrary predicates (Section 4.2).

In contrast to these previous approaches, our techniques operate *on the fly* and focus on the data, with very little upfront query analysis. This lets us support complex multi-query optimization in an extremely dynamic environment. Beyond streaming systems,

*This work was funded in part by the NSF under ITR grants IIS-0086057 and SI-0122599, and by the UC MICRO program.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

our fragment sharing approach can be used in traditional static databases, in order to share aggregate queries that have different predicates. We now summarize, our main contributions:

1. **Shared Time Slices (STS).** A new technique to share the processing of aggregates with varying windows that chops the input stream into time slices. We prove that our new *paired window* technique is an optimal method to form such slices. (Section 3)
2. **Shared Data Fragments (SDF).** A novel approach to share the processing of aggregates with varying arbitrary selection predicates that breaks the input data into disjoint sets of tuples. (Section 4)
3. **Shared Data Shards (SDS).** An innovative way to combine STS and SDF to share aggregates with varying windows *and* predicates. We know of no other approach to sharing aggregates that supports more than one kind of variation in the queries. (Section 5)
4. **On-the-fly MQO.** A common feature of all our techniques is that they operate on data on-the-fly and require no static analysis of the queries as a whole. This is vital in a dynamic streaming system, and very promising in static systems.
5. **Performance study.** We validate our approach with a study of an implementation of these techniques that evaluates their performance with a real data set. (Section 6).

We first present necessary background in Section 2 below.

2. BACKGROUND AND RELATED WORK

In this paper we consider *distributive* (e.g., `max,min,sum, and count`) and *algebraic* (e.g., `avg`) aggregates as classified by Gray et al. [15]. These are typical aggregates in database systems, and can be evaluated with constant state independent of the size of their input. Further, they can be computed using *partial aggregates* over disjoint partitions of their input,[1] a technique used with parallel databases (e.g., Bubba [3]), sensor networks (e.g., TAG [24]) and streams (e.g., STREAM [1], PSoup [5]).

2.1 Windowed Aggregation

Since data streams are unbounded, aggregates over them generally have a *window* specification. In CQL[2], a window is specified with a `RANGE`, and an optional `SLIDE` clause. For example, Query 1 computes the total value of high volume trades in "12 minute" windows (range), reporting these results at "5 minute" intervals (slide).

Query 1 *Total value of high volume xacts*

```
SELECT  sum(T.price * T.volume)
FROM    Trades T [RANGE '12 min' SLIDE '5 min']
WHERE   T.volume > 50000
```

We call a window with a slide *periodic*, and one without a slide *non-periodic*, or *on-demand*. A periodic window can be classified based on its range r and slide s as follows:

1. *Hopping:* when $r < s$, each window is disjoint.
2. *Tumbling:* when $r = s$, the windows are disjoint and cover the entire input.
3. *Overlapping:* when $r > s$, each window overlaps with some others.

We focus on queries that compute aggregates over periodic overlapping windows of streams, such as Query 1. With non-overlapping (hopping or tumbling) windows, such aggregates are easily computed and only require constant space, as a tuple can be discarded after being accumulated in the aggregate. In contrast, with overlapping windows a tuple is included in multiple windows and cannot be discarded in this way. Our sharing techniques are easily extended to non-overlapping windows.

In our implementation, we assume that an aggregate operator has heartbeats in its input so that it can produce results even if its input rate drops, or if there is a very selective predicate upstream [16]. Golab et al. [13] present a general treatment of this area.

2.2 Shared query processing

Shared aggregate processing has been studied in the MQO and materialized view literature for static databases. A big focus of this work has been for OLAP environments where queries differ in their grouping parameters. Harinarayan et al. [18] show how to efficiently compute data cubes by computing coarse grained aggregates from fine grained partial aggregates. Deshpande et al. [10] show a similar approach to share multi-dimensional aggregates using fine-grained chunks. The chunks approach also allows variation in predicates, but only when the predicates are exclusively over the grouping attributes. The work on chunks and data cubes was extended by Srivastava et al. [30] for a memory limited stream processor such as Gigascope [8], where the queries can only differ in their grouping parameters.

In streaming systems, Arasu et al. [1] explored shared processing of aggregates with varying non-periodic windows. While this work lets users specify different range intervals, it cannot exploit the cases where users are ready for results to be pushed to them periodically, and thus incurs heavy space and time overheads. Further, a shared aggregate that produces results on demand is not suitable for a streaming view whose results are used in other queries, since its on-demand nature forces it to be only used as the final operator in a string of queries. In fact, a composition of multiple queries is likely to get more specialized downstream (especially with selections) where there are fewer sharing opportunities.

Shared processing of queries with varying selection predicates and join windows has been studied extensively in streaming systems. Hammad et al. [17] showed how to share queries that have varying join window specifications. Work that focuses on sharing predicate evaluation in the presence of joins include NiagaraCQ [7], CACQ [23], and TULIP [21]. Although these papers study variations in windows and predicates, none of them permit aggregates, which is vital in processing high volume streams like stock trades.

Thus, the major open problems for shared processing of streaming aggregates is for queries with varying *arbitrary* predicates, *and* with varying periodic windows. The former case is open for static databases as well. This paper fills these lacunae in the literature.

3. VARYING TIME WINDOWS

In this section we address the first phase of our problem, i.e. sharing aggregates with varying windows. We develop *Shared Time Slices* (STS), a new technique for this case. We will use STS in Section 5 in solving our main problem of sharing aggregates with varying windows *and* predicates. STS can offer over an order of magnitude improvement over the state of the art unshared techniques. The queries that we consider here have identical predicates, and differ only in their periodic time windows, like Query 2 below.

Query 2 *Total sliding window transaction value*

```
SELECT   sum(T.price * T.volume)
FROM     Trades T [RANGE 'r' SLIDE 's']
```

Our approach chops an input stream into non-overlapping slices of tuples, that can be combined to form partial aggregates, which can in turn be aggregated to answer each query. We first explain

[1]The functions used for the partial aggregates can, in general, be different from those for the overall aggregate.

how to slice a stream for a single query using our new technique, *paired windows*, that is a significant improvement over earlier work. Next we show how to combine paired windows of multiple queries to produce partial aggregates over slices of the input that can still answer each query. Finally, we show why it is not feasible to combine paired windows statically, and instead present a *Slice Manager* that accomplishes this on the fly.

3.1 Slicing a stream with paired windows

Here we show how to efficiently process a single aggregate query with a periodic overlapping window. The idea of chopping a stream into slices was first introduced by Li [22] in the *paned window* approach where all the slices are of equal sizes and are called "panes". We improve on this with *paired windows* which chops a stream into pairs of possibly unequal slices. The paired windows technique is superior to paned windows as it can never lead to more slices. While both paned and paired windows can be used in our shared slices approach for processing multiple aggregates, we prove in Section 3.2 that paired window always produces better, or at least as good, sharing than paned windows. Paned and paired windows are both special cases of what we call non-overlapping sliced windows. We now define overlapping, and non-overlapping sliced windows.

Definition 1 (Overlapping) *An overlapping window W with range r and slide s ($r > s$) is denoted by $W[r, s]$ and is defined at time t, as the tuples in the interval:*

$$\begin{cases} [t-r, t] & \text{if } t \bmod s = 0, \\ \phi & \text{otherwise.} \end{cases}$$

Definition 2 (Sliced) *A sliced window W that has m slices is denoted by $W(s_1, \ldots, s_m)$. We say that W has $|W|$ slices, a period $s = s_1 + \cdots + s_m$, and that each slice s_i has an edge $e_i = s_1 + \cdots + s_i$. At time t, W is the tuples in the interval:*

$$\begin{cases} [t-s_i, t] & \text{if } t \bmod s = e_i,\ 1 \le i \le m \\ \phi & \text{otherwise.} \end{cases}$$

Intuition. An aggregate over an overlapping window $W[r, s]$ can always be computed by a two-step process that aggregates partial aggregates over a sliced window $V(s_1, \ldots, s_k, \ldots, s_m)$ with period s, if and only if, $s_k + \cdots + s_m = r \bmod s$. These sliced windows can be *paned* or *paired*, defined as:

1. *Paned:* $X(g, .., g)$; g is greatest common divisor of r and s.
2. *Paired:* $Y(s_1, s_2)$; $s_2 = r \bmod s$ and $s_1 = s - s_2$.

This intuition is based on the following Lemma. We omit the proof due to space constraints.

Lemma 1 *An aggregate over a window $W[r, s]$ can be computed from partial aggregates of a window $V(s_1, \ldots, s_k, \ldots, s_n)$ with period s if and only if:*

$$s_k + \cdots + s_n = r \bmod s$$

While paned windows break a window of period s into s/g slices of equal size g, paired windows split a window into a "pair" of exactly two unequal slices. Let A be the aggregate function we are computing over $W[r, s]$. The *partial* aggregation step uses a function G, and the *final* aggregation step uses a function H.[2] In the partial aggregation step, we apply the function G over all the tuples in each slice of V. In the final aggregation step, an overlapping window aggregate operator buffers these partial aggregates (that we call "sliced aggregates") and successively applies the function H on these sets of sliced aggregates to compute the query results. For example, Figure 1 shows how an aggregate over a window $W[18, 15]$ can use $X(3, 3, 3, 3, 3)$, a paned window of 5 slices, or a paired window $Y(12, 3)$.

[2]Since A is distributive or algebraic, G and H always exist. For example, if A is `count`, then G is `count` and H is `sum`.

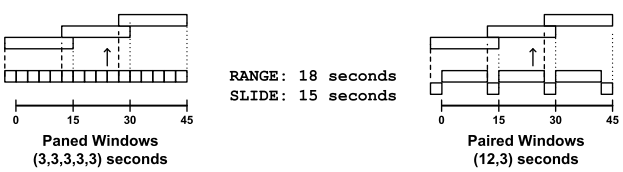

Figure 1: Paned vs Paired Windows

We now analyze the relative costs of executing a single sliced window aggregate using the paired and paned window approaches. Our analysis focuses on the costs of partial and final aggregation. Let T denote the set of tuples in each period s of the overlapping window $W[r, s]$. We measure costs in terms of the number of aggregate operations needed to process the tuples in T and summarize it for paned windows, and worst-case paired windows in Table 2.

Technique	Partial	Final
Paned	$\|T\|$	$(1/g)r$
Paired (worst-case)	$\|T\|$	$\lceil(2/s)r\rceil$

Table 2: Complexity of Paned and Paired Windows

In both cases the partial aggregate step requires $|T|$ operations. The cost of final aggregation depends on the number of partial aggregates, and so the number of slices, in a window period (its slide). If there are m such slices in a period s, the number of final aggregations in each period, i.e., the number of partial aggregates that are buffered, is $m(r/s)$. In a period s, paned windows always have s/g slices while paired windows have either 2 slices (worst-case when $r \bmod s \neq 0$) or 1 slice. This results in a final aggregate cost of $(s/g)(r/s)$ for paned windows, and $\lceil 2r/s \rceil$ for paired windows in the worst case (and r/s in the best case). Since the paired window option never has more slices than paned windows, it is always faster than, or at least as fast as, paned windows. For the rest of the paper we focus on paired sliced windows. We will also compare the performance of paired and paned windows in Section 6.

3.2 Combining multiple sliced windows

Here we show how to combine the paired, or paned, sliced windows of a set of queries in order to efficiently answer each individual query. We will prove that the paired window approach is the optimal way of using non-overlapping sliced aggregates to process multiple aggregate queries with differing periodic windows.

We start with $\mathbb{Q}$, a set of n queries that compute the same aggregate function over a common input stream, where each query has different range and slide parameters. More precisely, each query Q_i in $\mathbb{Q}$ has range r_i and slide s_i. These queries can be like Query 2, with different values for r and s, even where r and s are relatively prime. For simplicity we further assume that for all i, $r_i \bmod s_i \neq 0$ (the worst case for paired windows).

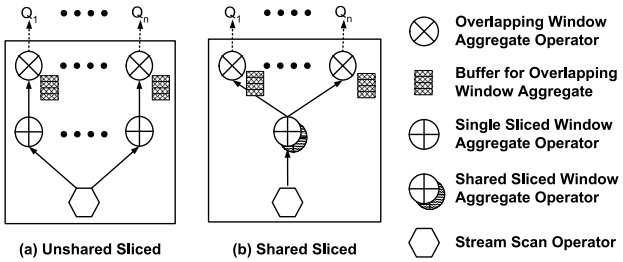

Figure 2: Possible plans for multiple queries

The queries in $\mathbb{Q}$ can be processed in either an unshared or shared

fashion. We consider each alternative in turn:

1. **Unshared sliced aggregation:** We process each query separately using paired windows, as shown in the query plan in Figure 2(a). The stream scan input is replicated to each of the n operator chains with a single sliced window aggregate that produces partial aggregates, which are fed to an overlapping window aggregate operator.
2. **Shared sliced aggregation:** We *compose* the paired windows of each query in $\mathbb{Q}$ into a single common sliced window (details below). Figure 2(b) shows the input stream processed by a *shared sliced window aggregate* producing a stream of partial aggregates, that is replicated to the n overlapping window aggregates.

3.2.1 Explicit common sliced window composition

We now show how to compose multiple sliced windows of the queries in $\mathbb{Q}$ to form a common sliced window. Partial aggregates computed with the common sliced window can then be used to answer each individual aggregate query. The main requirement is that the partial aggregates over the common sliced window must be computed at *every* unique slice edge of each individual sliced window.

Sliced windows can be composed only if they have the same period. Thus the period of a composite sliced window is the *lowest common multiple (lcm)* of the periods (or slides) of individual windows. With unequal periods, windows are *stretched* to the common period (*lcm*) by repeating their slice vectors. For example, Figure 3 shows how to compose two sliced windows $U(12,3)$ and $V(6,3)$. Here U and V have differing periods (15 and 9), and we *stretch* them respectively by factors of 3 and 5 to produce $U^3(12,3,12,3,12,3)$ and $V^5(6,3,6,3,6,3,6,3,6,3)$. We then *compose* U^3 and U^5 to produce a new composite sliced window $W(6,3,3,3,3,6,3,3,3,3,6,3)$. Note that ovals show shared edges in U^3 and V^5.

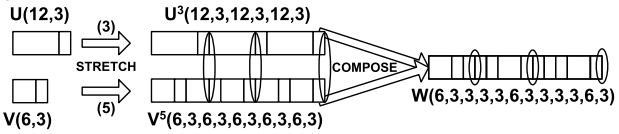

Figure 3: Composing sliced windows

We can now analyze the relative complexities of the unshared and shared approaches in processing the queries in $\mathbb{Q}$. Let S represent the composite period (*lcm*) of $\{s_1, \ldots, s_n\}$, the slides of the queries in $\mathbb{Q}$, and let T be the set of input tuples processed in the composite period S. Let E represent the number of slices, and the number of partial aggregates, formed in the common sliced window with period S. The partial aggregation step costs $n|T|$ for unshared sliced, and $|T|$ for shared sliced aggregation. In the unshared sliced approach, the cost of the final aggregation step for query Q_i at each period s_i is, $\lceil 2r_i/s_i \rceil$ from Table 2. Over a composite period S, there are S/s_i steps, causing a per-query cost of $(S/s_i)\lceil 2r/s_i \rceil$. In the shared sliced case, the cost of the final aggregation cost for the query Q_i at each period s_i is $(E/S)s_i\lceil r_i/s_i \rceil$, leading to the total per-query cost of $(S/s_i)(E/S)r_i$ over the composite period S. These costs are summarized in Table 3 below.

3.2.2 To share, or not to share

While the total final aggregation cost without sharing is always less than that with sharing, the total partial aggregation cost in the unshared case is always more than in the shared case. Let λ represent the input data rate and γ represent the rate at which partial aggregates are produced by the shared sliced window. Over a composite period S, λ is $|T|/S$ and γ is E/S. We can solve for λ and

Technique	Partial	Final
Unshared sliced	$n\|T\|$	$\sum_i (S/s_i)\lceil 2r_i/s_i \rceil$
Shared sliced	$\|T\|$	$\sum_i E(r_i/s_i)$

Table 3: Unshared versus Shared Aggregation

say that the shared approach is always better, as long as the input rate λ is high enough, as required by the following inequality (1).

$$\lambda > (\gamma \Sigma_i r_i/s_i - 2\Sigma_i r_i/({s_i}^2))/(n-1) \quad (1)$$

The critical factor is the "extent" of sharing in the common sliced window, and is reflected by the common partial aggregate data rate γ. In theory, with a very low input rate it may be better not to share. In practice, however, low input rates are unlikely. When they do occur, the shared and unshared approaches both cost so little, that the choice does not matter.

As an example, let us consider our motivating example that uses stock market data. We obtained a data trace of trades on the NYSE and NASDAQ for December 1,2004 from the NYSE TAQ [27] and NASTRAQ [26] databases. For this trading data, we found that the input rate λ is approximately 375 tuples a second. For the query workloads that we consider where the slide varies between 5 and 10 minutes and the range between 10 and 15 minutes, it is always better to share. We analyze this workload assuming a constant real data rate of 375 tuples/second and an artifical data rate of 0.0005 tuples/sec in Figure 4. Clearly, for the real data rate, the shared approach heavily outperforms the unshared approach. For an artificial data rate that is very low ($\approx$ 1.8 tuple/hour), the unshared approach is better for 83 or more queries. These simulation results are confirmed with time measurements from a real implementation that are presented in the performance study in Section 6.

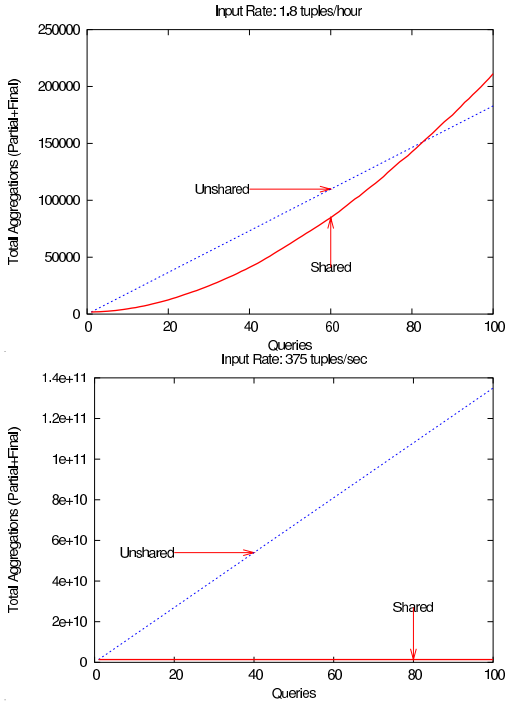

Figure 4: Analysis: Shared vs Unshared

3.2.3 On the optimality of shared paired windows

While we already know that paired windows outperform paned windows for individual queries, we now show that shared sliced

aggregation is faster using paired windows as compared to paned windows. In Theorem 1 we show that composing paired windows leads to a lower number of edges E, in a composite period S than composing paned, or any other sliced, window. We know from Table 3 that a smaller value for E lowers the cost of shared sliced aggregation, and so using paired windows is always optimal.

Theorem 1 *Let $\mathbb{W}$ be $\{W_1(r_1, s), \ldots, W_n(r_n, s)\}$, a set of n windows. Let W be the common sliced window formed by composing the paired windows of each W_i in $\mathbb{W}$. There exists no common sliced window W' formed by composing any other sliced window of each W_i in $\mathbb{W}$, where $|W'| < |W|$.*

Proof: Without loss of generality, let each W_i have identical slide s (or else stretch as in Section 3.2). So every sliced window of each W_i has edges at 0 and s. The paired window for each W_i has only one other edge at $s - r_i \bmod s$. From Lemma 1 every sliced window of each W_i must *also* have an edge at $s - r_i \bmod s$. Thus, the edges of the paired windows for each W_i *must* exist in all possible sliced windows of W_i. Since the edges of a composite sliced window are the union of all edges of its constituents, any composition W' of arbitrary sliced windows *must* include every edge of W, the composition of paired-window rewritings and $|W| \leq |W'|$. □

For the rest of this paper, unless mentioned otherwise, we always refer to paired windows when we talk about sliced aggregation.

3.3 On-the-fly sliced window composition

In Section 3.2 we saw how to explicitly compose sliced windows to form a common sliced window that can be used to efficiently process all queries in $\mathbb{Q}$. While composing sliced windows is conceptually simple, the resulting common sliced window can have a very long period (*lcm* of individual periods) with a large number of slices. Even tens of queries each with periods under 100 seconds can produce a composite sliced window with a period of 10^6 seconds and hundreds of thousands of slices. Such a window with a large slice vector consumes a lot of space and is expensive to compute. Here we present an elegant alternative, that produces a stream of partial aggregates for the common sliced window "on-the-fly" without explicit composition.

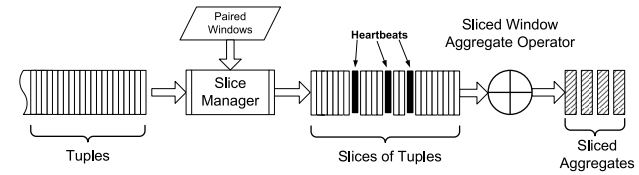

Figure 5: Slice Manager: Partial Aggregates

In our approach, we have a *Slice Manager* that keeps track of time and determines when to end the next slice, i.e., the time of the next slice edge. Figure 5 shows the Slice Manager demarcating the end of each slice with a *heartbeat* tuple. This heartbeat is a signal for the downstream sliced window aggregate operator, that it is time to emit partial sliced aggregates. This approach is very similar to the well-known sorting strategy for grouped aggregation [14].

The pseudocode for the core routines of the Slice Manager are shown in Algorithm 1. For simplicity we assume that each individual window $W_i(a, b)$ is a paired window, although the technique is easily extended to arbitrary sliced windows. Our algorithm is initialized with a set of paired windows, $W_1, \ldots, W_n$ by calling *addEdges* to add the edges of the first slice of each paired window to a priority queue H with operations (enqueue,dequeue and peek). Each edge identifies its time, the window it belongs to, and a boolean that records if it is the last slice in the window. The queue is ordered by increasing edge times. The SliceManager repeatedly calls *advanceWindowGetNextEdge* which returns the time of the next slice edge. At each call, this function discards all edges with the same time and at the top of the queue. If any edge belongs to the last slice of a window, it calls *addEdges* to add another set of edges for it. The Slice Manager passively outputs each input tuple, except when it receives a tuple with a timestamp greater than that of the next slice edge. At this point, it first inserts a heartbeat tuple in the output stream. When a query leaves the system, all edges corresponding to it are removed from the priority queue of the Slice Manager. Similarly, when a new query joins the system, appropriate edges are created for it by calling *addEdges* with the current time and the new paired window as an argument.

Algorithm 1 Slice Manager

```
proc addEdges(H, t_s, W(a,b))
  enqueue(H, edge(t_s + a, W, false));
  enqueue(H, edge(t_s + a + b, W, true));
end
proc initializeWindowState({W_1, ..., W_n})
  initializePriorityQueue(H);
  for i := 1 to n
      addEdges(H, 0, W_i);
  end
end
proc advanceWindowGetNextEdge(H)
  comment: Discard all edges at current time t_c.
  comment: Add new edges of subsequent periods.
  var Edge  e_c;
  var Time  t_c ← peek(H).time;
  while (t_c == peek(H).time)
        e_c ← dequeue(H);
        if (e_c.last == true)
          then addEdges(H, e_c.time, e_c.window);
        fi
  end
  return t_c;
end
```

To summarize this section, we showed how to efficiently process a set of streaming aggregate queries with identical selection predicates but varying periodic windows. A key feature of our implementation is that it does not require any static analysis of the query set, and can easily accomodate the addition and removal of queries.

4. VARYING SELECTION PREDICATES

In this section, we shift our focus to the other half of our problem, i.e., sharing the processing of aggregates with varying predicates. We develop *Shared Data Fragments* (SDF), a novel technique for this problem. Later, in Section 5, we will combine SDF with STS, our solution to the first half of the problem, to achieve our goal of shared processing of aggregate queries with varying windows and predicates. Our SDF technique can offer up to an order of magnitude improvement over the state of the art unshared techniques.

We begin with a precise problem formulation. Next, we present the intuition for shared fragments. Finally, we explain our novel on-the-fly scheme that obviates the need for static analysis of queries.

4.1 Problem Statement

We start with $\mathbb{Q}$, a set of n streaming aggregate queries that each compute the same aggregate function with an identical sliding window over an input stream where each query applies an arbitrarily

complex selection predicate. The predicate can include conjuncts, disjuncts, and even complex user-defined predicates. We say that the query Q_i in $\mathbb{Q}$ has a complex predicate $p(Q_i)$ (abbreviated to p_i) and an ungrouped aggregate A. For simplicity, we assume that each query Q_i has a tumbling window W (i.e., where the `RANGE` and `SLIDE` parameters are the same) and is similar to Query 3.

Query 3 *Total value of low volume mid-cap xacts*

```
SELECT  sum(T.price * T.volume)
FROM    Trades T [RANGE '5 min' SLIDE '5 min']
WHERE   (T.volume > 100) AND midcap(T.symbol)
```

Let W split the input stream into contiguous sets of tuples, and let T denote such a set. For ease of exposition, we focus on aggregation for a single set of tuples T for the rest of this section. Since we consider tumbling windows here, we merely have to apply our techniques for each subsequent set of tuples. We represent the subset of T that satisfies p_i by $p_i(T)$. Thus, we need to compute for each query Q_i, the aggregate $A(p_i(T))$ that we denote by $A_i(T)$ over the set of tuples T.

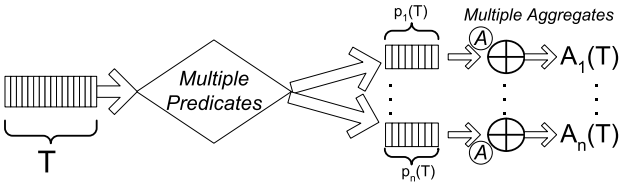

Figure 6: Unshared Aggregation

In many state-of-the-art systems such as TelegraphCQ [6] or STREAM [25], the tuples in T are processed by evaluating the predicates of all queries over each tuple in T. For many kinds of selections, systems like TelegraphCQ [20] and NiagaraCQ [7] build an index of the predicates to efficiently process tuples. After these predicates are processed, however, the input set T is split into n subsets that are each aggregated separately, a process we call *unshared aggregation* (Figure 6). Our goal in this section, is to reduce the total number of aggregate operations, and hence the associated cost, in evaluating such queries.

4.2 The Intuition

The main intuition is to use the predicates $\{p_1, \ldots, p_n\}$ to partition the tuples in a window of the input stream into disjoint subsets that we call *fragments*. The tuples in each fragment can then be aggregated to form partial *fragment aggregates* which can in turn be processed (via another aggregation) to produce the results for the various queries. In other words, a set of tuples T in a window of the input stream, is partitioned into $\{F_0, F_1, \ldots, F_k\}$, a set of $k+1$ disjoint fragments:

$$T = F_0 \cup F_1 \cup F_2 \cdots F_k$$

Each fragment F_i is associated with a subset of the query set $\mathbb{Q}$ that we denote by $Q(F_i) \in 2^{\mathbb{Q}}$, where *every* tuple in the fragment F_i satisfies the predicates of *every* query in $Q(F_i)$, and no other query. Our convention is that F_0 is a special fragment whose associated query set $Q(F_0)$ is the empty set ϕ. The tuples in F_0 satisfy none of the predicates $\{p_1, \ldots, p_n\}$, and thus do not need to participate in the aggregates of any query and can safely be ignored. Formally, each fragment F_i is created by applying the predicates $p_1, \ldots, p_n$ on tuples in T:

$$F_i = \{t \mid (t \in p(q) \forall q \in Q(F_i)) \wedge (t \notin p(q) \forall q \notin Q(F_i))\}$$

Example 1 *Consider a set of 3 queries Q_1,Q_2, and Q_3 with predicates p_1, p_2 and p_3. These result in a set of 8 possible fragments $\{F_0, \ldots, F_7\}$ with signatures as shown in Figure 7.*

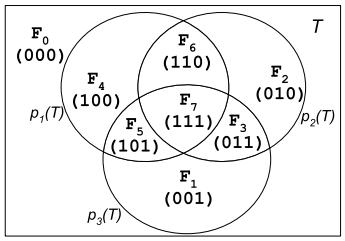

Figure 7: All possible fragments

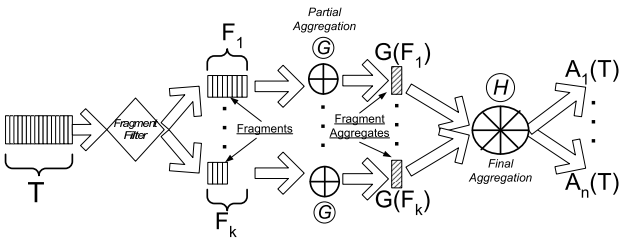

Figure 8: Conceptual View of Shared Fragments

Once we partition the set of tuples T into fragments we can efficiently compute the individual aggregates as shown conceptually in Figure 8. The basic idea is to split the aggregation process into two steps. First, in the partial aggregation step, we aggregate the raw data in each fragment F_i to produce a fragment aggregate with the value $G(F_i)$ and denoted by G_i. Then, in the final aggregation step, we aggregate these fragment aggregates to produce each query's results. As in Section 3.1, we denote the aggregation functions used in the partial and final steps by G and H respectively. Thus $A_i(T)$, the result of the aggregate query Q_i is given by:

$$A_i(T) = H\{G(F_j) \mid \forall F_j, Q_i \in Q(F_j)\} \quad (2)$$

For the queries in Example 1, each aggregate can be computed from the fragments $\{F_0, \ldots, F_7\}$ as follows:

$$A_1(T) = A(p_1(T)) = H\{G(F_4), G(F_5), G(F_6), G(F_7)\}$$
$$A_2(T) = A(p_2(T)) = H\{G(F_2), G(F_3), G(F_6), G(F_7)\}$$
$$A_3(T) = A(p_3(T)) = H\{G(F_1), G(F_3), G(F_5), G(F_7)\}$$

Our approach uses a dynamic implementation of this conceptual notion of shared fragments. One can, however, conceive of a more traditional static implementation in the vein of other MQO work. We now explain why such a hypothetical approach is not suitable for dynamic streaming environments.

A static approach would use *a priori* analysis of the fixed set of n queries in $\mathbb{Q}$, to determine which of the 2^n possible fragments can actually occur. This, however, would involve determining the subsumption relationships between the predicates of various queries, which is known to be computationally expensive [19]. Further, the system would need to manage this set of possible fragments, and for each tuple, efficiently compute which fragment it actually belongs to. Since there can be an exponential number of fragments, managing them can be expensive.

Even if it were possible to statically analyze the queries, the high cost of fragment management is difficult to ameliorate. This is be-

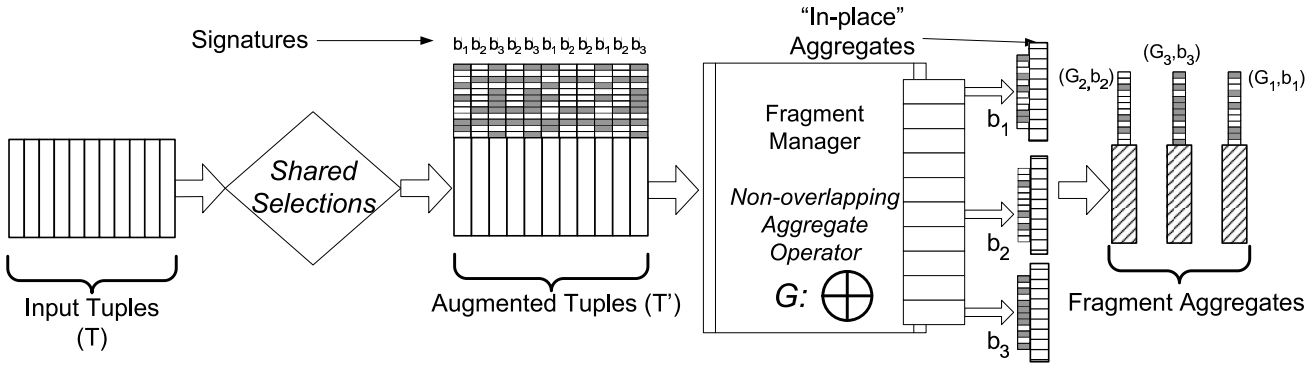

Figure 9: Dynamic Shared Fragments: Partial Aggregation

cause a static technique may not reveal a tight upper bound on the number of fragments, and will hence have high overheads. Static techniques can make pessimistic estimates of the number of fragments for the following reasons:

1. The predicates that we target can be arbitrarily complex and include opaque user-defined functions that cannot, in general, be easily analyzed.
2. A static analysis may not have access to information such as functional dependencies, or correlations, between attributes and might overestimate the number of fragments. In streaming systems, these might even vary significantly over time. For instance, in our stock trading application, heavy price fluctuation may be more often accompanied by high volume trades when the market is falling, than when it is stable.
3. Most importantly, in a streaming context, the real number of fragments is actually bounded by $|T|$, the number of tuples in a window. This number is not fixed, especially when we consider in Section 5, aggregate queries with varying selections *and* windows.

Beyond the fact that a static analysis of possible fragments is hard, and not complete in general, it is very unsuitable for the dynamic requirements of a data stream processor. A single query joining or leaving the system, can greatly affect the fragment computations. Thus, while the conceptual model of shared fragments that we introduced here is useful, we believe that a traditional MQO-style static analysis is not suitable for a data streaming system, and inappropriate for a traditional system.

4.3 Dynamic Shared Fragment Aggregation

We now describe our dynamic implementation of Shared Data Fragments. The main insight is that we can use existing data streaming technology to efficiently, and dynamically, identify the fragment a tuple belongs to. Notice that since this approach is dynamic, it is entirely free of the pitfalls of static analysis listed above. Thus, this scheme is useful even in a traditional, non-streaming, MQO context. We next explain in detail, the partial and final aggregation steps of this approach, and then present an analysis of the relative costs of unshared and shared aggregation.

4.3.1 Partial Aggregation

In our approach, we rely on the evaluation of selections in a scheme like CACQ [23] and TULIP[21] to produce an "augmented" stream of tuples, where each tuple carries along a signature that identifies the precise subset of queries that the associated tuple satisfies. While these selections may actually be applied in a shared fashion (e.g., the Grouped Selection Filter in CACQ and TULIP), sharing selections is not a requirement for our approach. Thus, given a set of tuples T, we can apply the predicates $\{p_1, \ldots, p_n\}$ on each tuple t_i in T to produce its signature b_i, and generate the augmented set T' of pairs of the form (t_i, b_i).

In CACQ and TULIP, the signature of a tuple is called its lineage, and is implemented with a bitmap containing one bit for each of the n queries in $\mathbb{Q}$. We use the same approach in our implementation. Since the signature of a tuple *encodes* the queries that it satisfies, it also identifies the unique fragment that the tuple belongs to, as well as the queries associated with that fragment. We represent the set of queries $Q(F_i)$ of a fragment F_i with a bit vector b_i that has n bits. We call b_i the *fragment signature* of F_i. Note that the j^{th} bit of b_i is set, if and only if, $Q_j \in Q(F_i)$. We denote the cardinality of $Q(F_i)$, i.e., the number of queries satisfied by the tuples in the fragment F_i, by $|b_i|$.

These augmented tuples are then processed by a *Fragment Manager* that dynamically combines, and aggregates, all tuples with identical signatures, i.e., those that belong to the same fragment. Given each tuple-signature pair (t_i, b_i), we look up the signature b_i in a data structure such as a hash table, or a trie, and accumulate t_i into an associated "in-place aggregate" using the partial aggregate function G. At the end of the partial aggregation step the fragment manager outputs a set of k fragment aggregate-signature pairs of the form (G_i, b_i), where G_i is $G(F_i)$, i.e., the result of applying the partial aggregation function G to the tuples in the fragment F_i.

This pipeline of partial aggregation in our dynamic shared fragment approach is shown in Figure 9. Notice that this is very similar to the well-known hash-based grouped aggregation [14] and we can easily adapt a standard operator implementation for our purpose.

4.3.2 Final Aggregation

In the final aggregation step (not shown in Figure 9), we combine $\{(G_1, b_1), \ldots, (G_k, b_k)\}$, the set of k fragment aggregate-signature pairs, as defined in equation (2). Algorithm 2 shows a straightforward techique for this step. We first initialize the final aggregate values $A_1, \ldots, A_n$ for each individual query. Then, we consider each fragment aggregate-signature pair in turn, and forward it to every query it is associated with, for aggregation. These queries are picked using the signature of each fragment aggregate.

4.3.3 Analysis

We now analyze the cost of processing the set of n selective aggregate queries with the unshared and shared techniques. As in Section 3, our analysis focuses on the computational complexity of aggregation operations in the partial and final aggregation steps.

We measure time complexity in terms of the number of aggregate operations carried out while processing the tuples in T. Ta-

Algorithm 2 Final Aggregation

```
proc FinalAggregation({(G_1, b_1) ..., (G_k, b_k)})
  for i := 1 to n, initialize(A_i); end
  for i := 1 to k
    for j := 1 to n
      if (b_i[j] = true) then A_j ← H(A_j, G_i);
    end
  end
```

ble 4 summarizes our analysis parameters. Each tuple $t \in T$ is augmented with a signature b to form the augmented set T'. Let k be the number of unique signatures in T', and let B be a set of k signature-frequency pairs $\{(b_1, f_1), \ldots, (b_k, f_k)\}$ where each pair (b_i, f_i) denotes that the signature b_i occurs f_i times in T'. We say that the expected cardinality of the signature of tuples in T' is α, and that the average cardinality of each signature in B is β.

$$\alpha = \frac{\sum_{(b,f)\in B} |b|f}{|T|} \qquad \beta = \frac{\sum_{(b,f)\in B} |b|}{k}$$

Symbol	Explanation
n	Number of queries
T	Set of tuples in the window
T'	Augmented set of tuples in the window
k	Number of unique signatures in T'
B	Set of k signature-frequency pairs in T'
α	Expected cardinality for each signature in T'
β	Average cardinality for each signature in B

Table 4: Parameters

Table 5 summarizes the costs of the unshared and shared techniques to process n selective aggregate queries. With unshared aggregation, there is only a single "final aggregation" step. Here, each tuple in T is subjected to as many aggregations as the number of queries it satisfies. Although this method only uses the input set T, its cost can be calculated by considering the augmented input set T' as follows:

$$\sum_{(t,b)\in T'} |b| \quad = \quad \sum_{(b,f)\in B} |b|f \quad = \quad |T|\alpha$$

With the shared approach, the partial aggregation step involves exactly $|T|$ aggregations and produces a set of k fragment aggregates each associated with a signature in B. As each fragment aggregate is aggregated as many times as the cardinality of its associated signature, the final cost is:

$$\sum_{(b,f)\in B} |b| = k\beta$$

Technique	Partial	Final
Unshared	0	$\alpha\|T\|$
Shared	$\|T\|$	$k\beta$

Table 5: Unshared and Shared Aggregation Costs

From this analysis, the shared approach is cheaper than the unshared approach when $k << |T|$, i.e., the number of fragment aggregates has to be significantly less the input data set. If this is not true, and $k \approx |T|$ (recall that k cannot be greater than T), then the expected cardinality of each signature in T' approaches the average cardinality for each signature in B, i.e., $\alpha \approx \beta$. In this situation, the cost of shared aggregation approaches $|T| + |T|\alpha$ which exceeds $|T|\alpha$, the cost of unshared aggregation.

Consider for instance, an example based on a scenario with 256 queries from our performance study (the scenario is explained in detail in Section 6.3) processed over one hour of 10 minute intervals of stock market data. Here, $|T|$, the average size of the input set for 10 minute intervals over one hour is 189,445. Let the number of fragments caused by our query workload be k. Also, let us assume for simplicity, that an identical number of tuples belong to each fragment, i.e., α is the same as β. From our workload we set $\alpha = \beta = 76$.

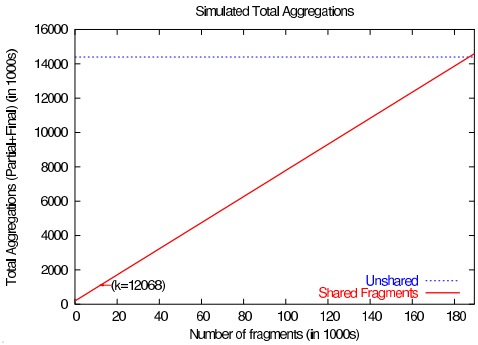

Figure 10: Analysis: Shared vs Unshared

Figure 10 shows the simulated costs of shared and unshared aggregation for this scenario, where the number of fragments k, varies from 1 to 189,445. In the actual workload, the average number of unique fragments was 12,068. The arrow in the figure shows the point that represents the costs of shared aggregation for this case.

While the maximum possible fragments is governed by the number and nature of the queries (especially the number of attributes they involve) in a workload, the actual number of fragments in a window is based on the nature of the data set. In this workload that uses real data, the number of fragments (k=12,068) is far smaller than the number of tuples in a window ($|T|$=189,445). In this particular case, there are fewer fragments because the values found in our real stock market data, are not uniformly distributed in their entire domain. For example, in the first 10 minute interval of the stock market data discussed above, the `volume` attribute of each trade tuple has values between 10 and 1.6 million. There are, however, fewer than 2000 distinct values seen for this attribute in roughly 200,000 tuples of the interval. Further, the most common value, 100, occurs for half the tuples, and the 15 most common values together account for over 90% of the tuples.

We expect that in practice, we will find this situation, i.e, the number of observed fragments being small relative to the number of tuples in a given window, true for many real applications.

In this section, we presented the Shared Data Fragments (SDF) approach to efficiently process multiple streaming aggregate queries with varying selections and identical windows. We proposed an innovative dynamic implementation of Shared Data Fragments that leverages existing advances in stream query processing without any of the drawbacks of static analysis. We present a detailed experimental evaluation of this approach in Section 6.

5. PUTTING IT ALL TOGETHER

In this section we are now ready to address our main problem of shared processing of aggregate queries with varying predicates, *and* windows. Our approach is to put together the techniques developed

in the earlier sections (STS in Section 3 and SDF in Section 4) that solve simpler versions of this problem. We show here that these techniques work well together to form our new technique, *Shared Data Shards* (SDS), and solve the main problem.

Here, we compare our SDS approach with *Unshared Sliced* (US), a scheme where each query has an operator chain with a sliced window aggregate followed by an overlapping window aggregate. For each input tuple, we can apply the predicates of the queries, possibly in shared fashion. Then, for each query the tuple satisfies, we replicate the tuple to its operator chain.

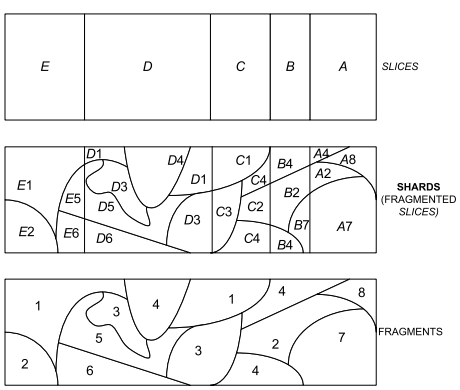

Figure 12: Shards

From a high-level, both STS and SDF partition an input data set into chunks of tuples that are partially aggregated, and then aggregate these partial aggregates to answer individual queries. Thus, when both windows and selections vary, we can conceive of an approach that partitions the input set to form what we call *shards* (these can be thought of as *fragmented slices* or *sliced fragments*) as shown in Figure 12, partially aggregating the tuples in each shard, and aggregating these partial aggregates to answer each query.

We cannot, however, further partition a partial aggregate formed by sliced or fragmented aggregation. Instead, we must first apply one partitioning operation (slicing or fragmentation) to the input stream, and then apply the other operation to these partitions *before* computing any partial aggregates. Thus, the main question is:*"which partitioning operation, slicing or fragmentation, should we perform first ?"*

To answer this question, we must look back to the implementations of STS and SDF that we presented earlier. Recall that while partial aggregate computation is similar to the sorting based strategy for grouped aggregates in STS, it is similar to a hashing based strategy for grouped aggregates in SDF. The reason that the sorting based strategy is possible for STS, is that in a data stream, tuples are naturally sorted by time. In contrast, contiguous tuples do not necessarily have identical signatures (and hence do not belong to identical fragments), and so SDF needs to use a hashing based strategy. The consequence of this observation is that, while the partial aggregation step is performed "in-place" along with partitioning step in SDF, it can safely be separated from the partitioning step in STS.

Using this insight, we propose the Shared Data Shards (SDS) technique that uses elements from the STS and SDF approaches. Like in STS, we use a Slice Manager that is aware of the paired windows of each query in the system, to demarcate slice edges in an input stream using heartbeats. These slices of tuples are then passed on to a shared selections operator (e.g., GSFilter), to produce slices of augmented tuples, just as in SDF. These augmented tuples are then sent to a SDF-style Fragment Manager that computes partial aggregates of shards. Next, these shard aggregates are processed using SDF-style Final Aggregation (Section 4.3.2) and sent to the appropriate per-query overlapping window aggregates. Finally, these operators produce result tuples for each query. This pipeline is shown in Figure 11.

An analysis of the relative complexities of the unshared (US) and shared (SDS) schemes for processing aggregate queries with varying selections and windows is essentially the same as that for the unshared and shared (SDF) scheme in Section 4.3.3. The main difference is that, unlike STS and SDF, the SDS approach ends up splitting the input data into much smaller shards. Even so, we will show in the next section that for the real data sets we used, such as the stock market trading data, the performance of SDS greatly exceeded that of US, as explained in the performance study in Section 6.

In this section, we presented Shared Data Shards (SDS), a new approach to efficiently share the processing of multiple streaming aggregate queries that differ in selections and periodic windows. This solves an important problem in understanding how to share aggregate queries that vary in more than one aspect. A key property of this scheme is that it achieves the objectives of shared processing, without requiring *any* prior analysis of the queries involved. This is important because of two reasons stated earlier: (1) the kinds of analysis required for sharing such queries is hard, and (2) in a data streaming context, queries are expected to join and leave at any time and system resources cannot be engaged in complex multi-query optimization while processing live data.

6. PERFORMANCE STUDY

In this section, we present a detailed performance study of the various approaches that we proposed in this paper. We use intra-day trading data from the NYSE and the NASDAQ, that corresponds to our examples throughout the paper. Our study focuses on each of the three problems that this paper addresses and is summarized in Table 6. We first describe our experimental setup and then present and analyze our results.

Workload	Predicates	Windows	Section
(A)	Same	Different	6.2
(B)	Different	Same	6.3
(C)	Different	Different	6.4

Table 6: Query Workloads

6.1 Experiment Setup

We built a prototype aggregate query processor for data streams in Java. With our prototype we can realize query plans for all the schemes described in this paper.

The data we use is summarized in Table 7 below, and consists of a stream (Trades) with intra-day trading data from the NYSE [27] and NASDAQ [26] stock markets on December 1st 2004 during the regular trading hours of 09:30 AM to 4:30 PM, a static table (Close) with the previous day's closing price for all stocks, and a static table (Index) that reflects whether or not a given stock is in any of the Russell 3000, Russell 2000, or Russell 1000 indexes.

Name	Schema	Type
Trades	(Time, Symbol, Price, Vol)	Stream
Close	(Symbol, CP)	Table
Index	(Symbol, R3000, R2000, R1000)	Table

Table 7: Data Schema

Each workload has a query set with {16,32,64,128,256} queries over one hour's worth of data starting at 12:00 noon. The queries

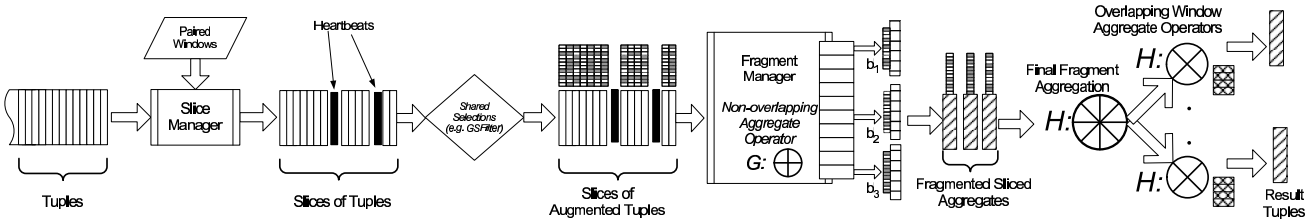

Figure 11: Shared Data Shards

are all based on the template in Query 4. They involve a join on the Symbol attribute between the Trades stream, and the Close and Index tables. Each query computes the total transaction value of all trades, subject to possible restrictions, in a sliding window. The condition checked could be whether or not the trade represented an "outlier" in terms of its volume, or whether or not it revealed a significant price movement of the equity from the previous day's closing price.

Query 4 *Query Template*

```
SELECT sum(T.price * T.vol)
FROM   Trades T[RANGE r SLIDE s],Close C,Index X
WHERE  T.Symbol = C.Symbol AND T.Symbol = X.Symbol
       [AND Member AND Value]
```

These restrictions are summarized in Table 8. Queries in (A) have a range and slide picked uniformly at random from [600,900] and [300,600] seconds respectively. Queries in (B) have a complex predicate with "Member" and "Value" conjuncts. The Member conjunct picks with uniform probability a market index from one of {R3000, R2000, R1000}, and with uniform probability checks whether or not the traded stock belongs to it. The Value conjunct picks with equal probability a quantity that is one of the volume, value, or % change for the day, and checks, with equal probability, if this quantity is greater than, or less than a constant. Note that queries in (A) have no predicates, and queries in (B) have the range and slide both set to 600 seconds. We explain the queries in (C) in Section 6.4.

Type	Name	Values
Window	Range: r Slide: s	[600,900] seconds [300,600] seconds
Predicate	Member	X.R3000: true, false X.R2000: true, false X.R1000: true, false
Predicate	Value	Vol: > V, < V Vol*Price: > W, < W abs(Price-CP)/CP: > F, < F

Table 8: Query Parameters

In this paper we focus on processing shared aggregates. We assume the periodic evaluation can be sped up by using well known techniques such as Rete [11], CACQ [23] and NiagaraCQ [7]. Thus, in our experiments we measure the actual time in processing the aggregates and any accompanying overheads, such as the use of hash-tables. In order to minimize any effects of I/O, we buffer our data in memory. Each value that we report for any workload with n queries is an average computed from 10 iterations each with a different set of n queries.

6.2 (A) Same Predicates, Diff. Windows

In this workload, we examine queries with identical selection predicates, and different periodic windows, over a real data set. We compare the execution time across 4 strategies (Unshared Paned, Unshared Paired, Shared Paired, Shared Paned) based on the unshared sliced and shared sliced approaches discussed in Section 3.

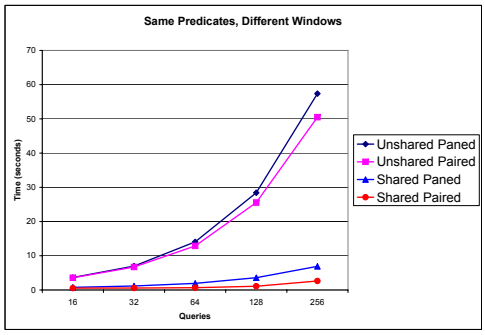

Figure 13: Sliced Aggregates

The results for all 4 strategies are shown in Figure 13. Here, the shared approaches significantly outperform the unshared schemes by more than an order of magnitude. For instance, with 256 queries, unshared paired costs 50.48 seconds, while shared paired costs only 2.63 seconds. This is because, although unshared paired fewer final aggregations (6813) compared to shared paired (1,462,391), unshared paired has a many more partial aggregations (291,490,816) compared to shared paired (1,138,636).

Further, for all query sizes, the paired approach outperforms the paned approach. For this 256 queries case, shared paned costs 6.90 seconds - more than twice as much as shared paired. This is because, shared paned has far more final aggregations (2,340,842), and has to buffer many more partial aggregate tuples.

These results match our analysis from Section 3.2. First this analysis correctly predicted (Section 3.2.2) that with high data rates the shared approaches will heavily outperform the unshared approaches. Second, we proved in Section 3.2.3 that paired windows are are better than paned windows, and in fact are optimal, for shared processing. Again, this matches our experimental results.

6.3 (B) Diff. Predicates, Same Windows

In this workload, we examine queries with differing selection predicates, and identical tumbling windows. Since these are tumbling windows, the paired and paned options are identical and there is no final overlapping window aggregate step. Thus, we compare the execution time for the Unshared and Shared Data Fragments strategies from Section 4.

The results are shown in Figure 14 with a split of the partial and final aggregation costs for the Shared Data Fragments scheme. For all query sizes, the Shared Data Fragments (SDF) approach vastly outperforms the state of the art Unshared Sliced (US) approach. For example, with 256 queries, the US scheme uses 27.25

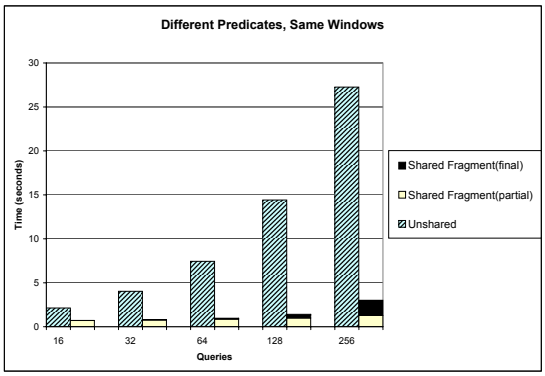

Figure 14: Unshared/Shared Data Fragments

seconds, while the SDF scheme uses only 3.01 seconds, a savings of 89% and almost an order of magnitude. In this case, US has to compute 116,101,112 aggregations, while SDF has to compute only 7,771,956 total aggregations (1,094,947 partial and 6,677,009 final). Note that although SDF has only 6% the number of aggregations as US, it has a total time about 11% of US. This is because of the overheads of SDF, such as hash table and bitmask operations.

To understand why the SDF scheme gives us such outstanding results, we focus on the number of tuples and average number of unique fragments. For instance, with 256 queries, the window from 12:20 to 12:30 has 218583 tuples, but only 11,944 unique fragments. In fact, over a 1 hour time period from 12:00 to 1:00, the average number of tuples per 10 minute window is 192,945 and the average number of fragments per window is 11,407. More interestingly, over the entire 1 hour period (with six 10 minute windows), there are 1,181,901 tuples, but only 34,327 unique fragments. Notice that while any static analysis scheme would have analyzed the query workload as splitting the data set into at least 34327 fragments, our dynamic approach gives us all the benefits of such a static approach while only having to manage on average 11,407, and at most 11,893 fragments.

In summary, we found that SDF can provide an order of magnitude improvement over the Unshared approach. The SDF scheme performs very well because of the small number of unique fragments that occur in each window.

6.4 (C) Diff. Predicates, Diff. Windows

We now consider workloads that represent our main problem, i.e., aggregate queries with varying predicates *and* windows. We consider in turn, a *regular* workload, and a *low sharing* workload, and for each case we compare the performance of the Shared Data Shards (SDS) with the Unshared Paired (from Section 5) approaches.

In the first *regular* case, we consider query sets with sizes in $\{16,36,64,144,256\}$. Here, a query set with n queries has predicates chosen from $\sqrt{n}$ distinct predicates (based on (B) above), and windows chosen from $\sqrt{n}$ distinct windows (based on (A) above). These sets are constructed so that all queries in a set of n queries are unique, but with exactly $\sqrt{n}$ unique predicates and windows. In the second *low sharing* case, we have sets of queries with sizes in $\{16,32,64,128,256\}$. Here, each query set is constructed by combining the properties of the sets used in (A) and (B) defined above. Thus, a given query set will have neither any duplicate windows, nor any duplicate predicates.

6.4.1 Regular Workload

The results for the regular workload are shown in Figure 15. We do not separate out the final aggregation cost in the UP scheme as it is negligible. For all query sizes, the SDS approach consistently outperforms the UP method. For instance, for 256 queries, the UP method costs 31.19 seconds, while the SDS approach costs only 3.67 seconds, a savings of 89%. Notice that here, SDS provides nearly an order of magnitude improvement over UP, for a workload where every query is different. This is because in this case, UP has to perform 100,909,915 (100,902,960 partial + 6,955 final) aggregations, while SDS only has to perform 3,395,132 (1,138,636 partial + 2,256,496 final) aggregations. Again, while SDS only performs about 3.3% of the aggregations of US, its cost is about 11% of US. The difference can be attributed to the overheads of SDS. Note that the difference here is about 7% and less than the 5% difference in the SDF case. This is because SDS has all the overheads of SDF, as well the additional overheads of the Slice Manager.

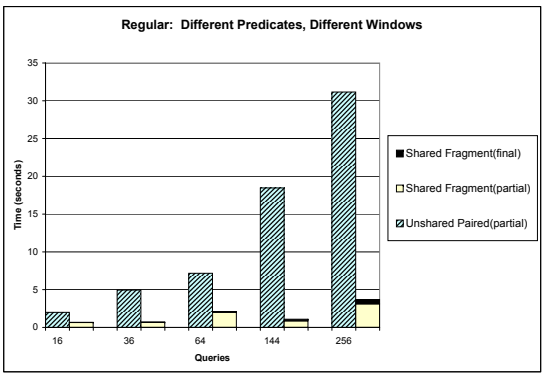

Figure 15: Regular: Unshared/Shared Data Shards

6.4.2 Low Sharing Workload

We now consider the low sharing workload where every query is unique, *and* every predicate and window is also unique across *all* queries.

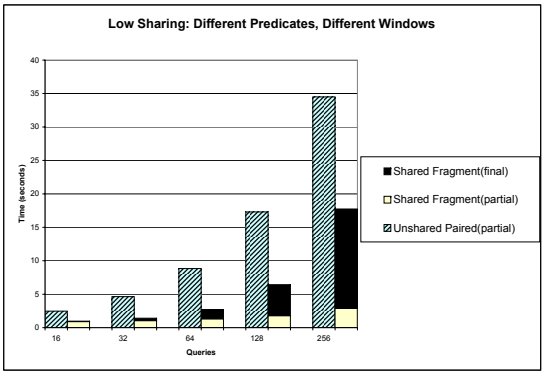

Figure 16: Low Sharing: Unshared/Shared Data Shards

This is a low sharing workload, where at first blush it might seem that there are no easy sharing opportunities. Even so, our results are very encouraging and are shown in Figure 16. The partial and final aggregation costs for SDS are split and we omit the final aggregation costs of UP. For all query sizes, SDS significantly outperforms UP. For instance, for 256 queries, the total cost for UP is 34.56 seconds, while that for SDS is only 17.73 seconds. This is a savings of approximately 51%, or a factor of 2. Again, these savings can be attributed to the fewer aggregations that SDS performs (65,278,107

total = 1,294,064 partial and 63,984,043 final) as compared to UP (120,679,022 total = 120,672,210 partial and 6,812 final).

Notice that this savings is less than the order of magnitude improvements we see in the regular workload above. Essentially, when both predicates and windows vary, the input stream gets partitioned into small "shards" (Figure 12). Since there is more variation in this workload, there are more shards, each with fewer tuples.

We emphasize that in such a low sharing workload where there are no repeated windows or predicates, the other schemes (STS and SDF) cannot be used here.

In summary, our experiments show that when both predicates and windows vary, the opportunities for sharing can be small or large. In either case, SDS can exploit these opportunities and provide improvements of between a factor of 2 (for fewer opportunities) and a factor of 10 (for greater opportunities) over UP.

6.5 Summary

Our study examined different ways to process sets of aggregate queries with varying predicates and windows. In all our experiments, our dynamic sharing approach gives large benefits over the state of the art. The specific conclusions we can make are:

1. **Paired beats Paned.** Paired windows are always superior to the paned windows, whether for a single query or for multiple queries where only windows vary. This result reinforces our proof on the optimality of paired windows in Section 3.2.
2. **Shared Data Fragments beats Unshared.** In our experiments, when only predicates vary we found that there are far fewer unique fragments than tuples in any given window. As a result, SDF offers up to an order of magnitude improvement over the unshared technique.
3. **Shared Data Shards beats Unshared Sliced.** In our experiments, we found that when predicates and windows *both* vary, the SDF approach can offer improvements of between a factor of 10 and a factor of 2 over UP. The latter case is for extremely aggressive stress tests where no window or predicate occurs in more than one query.

7. CONCLUSIONS

Data streaming systems are increasingly used as critical infrastructure for monitoring applications such as financial alerts. Such systems have to support hundreds of similar, but different, concurrent queries. Executing these queries separately can lead to excessive system load, and force the use of query admission control, or load shedding.

Instead, we showed how the processing of such queries can be shared, when they vary in their predicates and windows. First, for queries with differing windows, we developed "Shared Time Slices". Next, for queries with differing predicates, we proposed "Shared Data Fragments". Finally, we showed how these two techniques can be combined together in the "Shared Data Shards" approach, in order to solve our problem.

Our strategies mark a significant departure from the state of the art in shared processing, where queries are normally optimized together statically to produce an efficient execution plan. The traditional approach is not suitable in a streaming system where queries join and leave at any time. Further, we showed why the static approach is computationally expensive for queries with varying selections or varying windows. Instead, we proposed innovative ways to share the processing of queries on the fly by examining the data as it streams in, with very little upfront query analysis. Not only is this a very efficient scheme, it lets us elegantly handle environments with lots of *churn*, i.e., where queries come and go very often.

Finally, we evaluated an implementation of our approach with a data set based on stock market trades, and showed that our schemes can perform excellently in the real world.

8. REFERENCES

[1] A. Arasu *et al.*. Resource sharing in continuous sliding-window aggregates. In *VLDB*. 2004.

[2] A. Arasu, *et al.*. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, (To appear).

[3] F. Bancilhon, *et al.*. FAD, a powerful and simple database language. In *VLDB*. 1987.

[4] D. Carney, *et al.*. Monitoring streams - a new class of data management applications. In *VLDB*. 2002.

[5] S. Chandrasekaran *et al.*. Streaming queries over streaming data. In *VLDB*. 2002.

[6] S. Chandrasekaran, *et al.*. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*. 2003.

[7] J. Chen, *et al.*. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD*. 2000.

[8] C. D. Cranor, *et al.*. Gigascope: A stream database for network applications. In *SIGMOD*. 2003.

[9] M. Denny *et al.*. Predicate result range caching for continuous queries. In *SIGMOD*. 2005.

[10] P. M. Deshpande, *et al.*. Caching multidimensional queries using chunks. In *SIGMOD*. 1998.

[11] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object match problem. *Artifical Intelligence*, 19(1):17–37, September 1982.

[12] M. J. Franklin, *et al.*. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*. 2005.

[13] L. Golab *et al.*. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD*. 2005.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[15] J. Gray, *et al.*. Data Cube: a relational aggregation operator generalizing group-by, cross-tab and sub-total. In *ICDE*. February 1996.

[16] M. A. Hammad, *et al.*. Efficient pipelined execution of sliding window queries over data streams. Technical Report CSD TR#03-035, Purdue, 2003.

[17] M. A. Hammad, *et al.*. Scheduling for shared window joins over data streams. In *vldb*. 2003.

[18] V. Harinarayan, *et al.*. Implementing data cubes efficiently. In *SIGMOD*. 1996.

[19] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*. Springer Verlag, 1985.

[20] S. Krishnamurthy, *et al.*. TelegraphCQ: An architectural status report. *IEEE DE. Bull.*, 26(1), 2003.

[21] S. Krishnamurthy, *et al.*. The case for precision sharing. In *VLDB*. 2004.

[22] J. Li, *et al.*. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, March 2005.

[23] S. R. Madden, *et al.*. Continuously adaptive continuous queries over streams. In *SIGMOD*. 2002.

[24] S. R. Madden, *et al.*. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*. 2002.

[25] R. Motwani, *et al.*. Query processing, resource management, and approximation in a data stream management system. In *CIDR*. 2003.

[26] NASDAQ. NASTRAQ: North American Securities Tracking and Quantifying System. http://www.nastraq.com/description.htm.

[27] NYSE. NYSE TAQ: Daily Trades and Quotes Database. http://www.nysedata.com/info/productdetail.asp?dpbid=13.

[28] P. Roy, *et al.*. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*. 2000.

[29] T. K. Sellis. Multiple-query optimization. *ACM TODS*, March 1988.

[30] D. Srivastava, *et al.*. Multiple aggregations over data streams. In *SIGMOD*. 2005.

[31] N. Tatbul, *et al.*. Load shedding in a data stream manager. In *VLDB*. 2003.