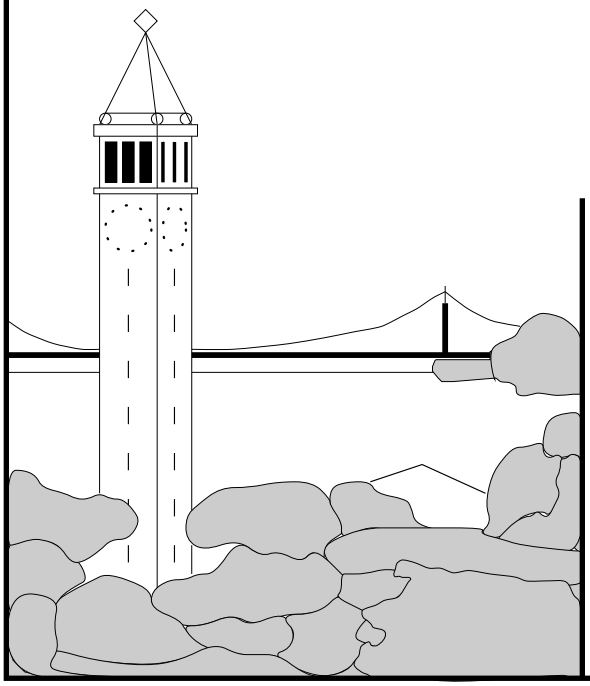
Shared Hierarchical Aggregation for Monitoring Distributed Streams

Sailesh Krishnamurthy *Michael J. Franklin*

Computer Science Division, University of California, Berkeley, CA 94720
{sailesh,franklin}@cs.berkeley.edu



Report No. UCB/CSD-05-1381

October 18, 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Shared Hierarchical Aggregation for Monitoring Distributed Streams

Sailesh Krishnamurthy and Michael J. Franklin

Computer Science Division, University of California, Berkeley, CA 94720
{sailesh,franklin}@cs.berkeley.edu

Abstract. Widely dispersed monitoring networks generate huge data volumes that are naturally organized via hierarchical aggregation. In a system that manages such data, applications pose periodic aggregate queries. In this paper we show how to efficiently process multiple periodic aggregate queries in a hierarchy. First, we use a novel query rewrite that optimally executes individual queries. Next, we show how to combine the rewritten queries to share computation and communication resources. Finally, we identify a challenge in shared aggregation across a heterogenous hierarchy, namely that push-down reduces sharing and pull-up increases communication. We then propose a "partial push-down" technique that permits effective sharing without increasing communication costs.

1 Introduction

Enterprises are deploying receptors such as network monitors (for intrusion detection) and RFID readers (for asset tracking) across wide geographies, driving the need for infrastructure to manage and process the data streams produced. We are building HiFi [4], a general purpose system to manage, process, and query data streams from widely dispersed receptors. Figure 1 shows distributed intrusion detection powered by HiFi where edge nodes monitor network traffic.

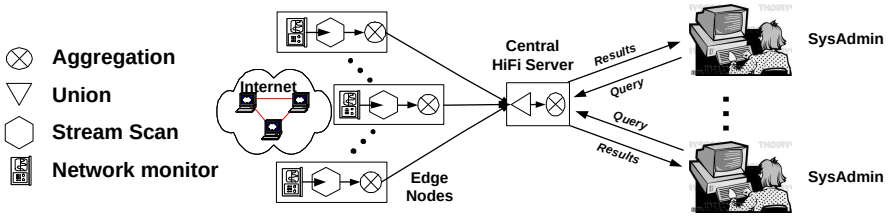


Fig. 1. Using HiFi for distributed intrusion detection

In large-scale monitoring applications there can be a large number of users executing similar concurrent queries. Using a naïveapproach to execute such queries can lead to scalability and performance problems as each additional query can add significant load to the system. Instead, it is vital to *share* computational and communication resources by exploiting the similarities in these queries. The techniques we present in this paper let a HiFi system exploit such sharing in order to support a large number of concurrent queries.

1.1 Challenges in Monitoring Distributed High-Volume Streams

In our intrusion detection example administrators can pose periodic windowed aggregate queries over network monitor data. Such queries have two parameters: a *range* which is an interest interval over which the aggregate is computed, and a *slide* which is a periodic (the slide is the period) interval controlling when results are reported. We consider HiFi deployments with the following properties:

1. *Periodic:* Reporting results at specific intervals avoids flooding the network with data. This is used in sensor databases and single site systems.
2. *Overlapping:* Large ranges and small slides cause overlapping aggregates. Tuples in many windows makes efficient execution challenging.
3. *Sharing:* Resource sharing is a common way to scale streaming systems to many queries.
4. *Hierarchical:* Bottom-up computation, across space and time, lets HiFi scale with streams from widely deployed receptors.

Challenge: Our goal is to efficiently process a large number of aggregate queries with periodic, overlapping windows over high volume data streams from widely distributed receptors that are organized in a hetergenous hierarchy.

While some of these properties have been addressed in other systems, we argue that all of them are *together* necessary in emerging large scale receptor-based systems like HiFi. We know of no other system that attempts to support all these properties. For instance, while TAG[12] supports periodic hierarchical aggregates, it does not support overlapping windows and sharing. Similarly, STREAM[1] shares non-periodic overlapping windows in a single-site system.

It turns out that these properties are mutually incompatible for plans that either push-down or pull-up aggregates. Aggregate pull-up moves raw data up the hierarchy and increases communication overheads. On the other hand, while aggressive aggregate push-down reduces the communication costs of individual queries, it lowers opportunities for sharing, leading to high overall computation and communication costs. This tension has important implications for distributed monitoring systems like HiFi. Bandwidth consumption in a wired network affects real operating costs with usage-based pricing. In a wireless sensor network it determines battery life. In contrast, computing capacity is plentiful in high end servers and sparse in sensors and edge nodes. Thus, it is vital to distribute the load across the HiFi resource hierarchy. We resolve this tension using a *partial push down* technique that has the following benefits:

1. Communication, and some computation, is shared across the hierarchy.
2. The remaining computation that cannot be shared is pulled up the hierarchy where resources are abundant.

Our partial push down technique is implemented as a series of operations on a set of queries as shown in Fig. 2. First, a new *paired window* rewrite extracts non-overlapping parts of each query. These non-overlapping parts are then *composed* to form a common sub-query. Finally, we pull-up the overlapping parts of each query and push-down the non-overlapping common sub-query.

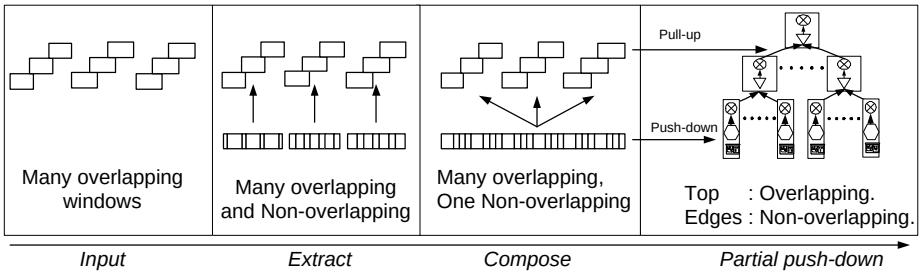


Fig. 2. High-level overview of partial push-down

Contributions. Efficient shared processing of aggregate queries in a hierarchy is hard because the standard push-down and pull-up approaches are both unsuitable. Thus, we need to invent techniques like new query rewrites and partial push-down in order to let a HiFisystem exploit sharing and support many concurrent queries. The specific contributions of this paper are the following:

1. A novel way to extract and execute non-overlapping components of an aggregate over an overlapping window. We prove that our *paired window* rewriting is optimal and superior to previous work in Sect. 3.
2. A technique that to share the processing of overlapping aggregates by composing non-overlapping paired windows. We also prove that this is an optimal way of sharing non-overlapping windows in Sect. 4
3. The partial push-down technique to share communication of non-overlapping aggregates in a hierarchy in Sect. 5.
4. A performance study with real data validating our approach in Sect. 6.
5. A principled way to overlay partial push down across a heterogenous hierarchy in Sect. 7.

We first present necessary background in Sect. 2 below.

2 Background and Related Work

2.1 Distributed Intrusion Detection

Our driving example is distributed intrusion detection in an enterprise. Here, each node at the edge of an enterprise monitors all the incoming network traffic (i.e., incoming TCP SYN packets) using a tool such as Snort [14]. The edge nodes stream this data to a higher level server. Figure 1 shows a two-level hierarchy that can be extended with additional levels if there are too many edges. Let the union of data from edges be a single virtual stream `Snort`, with attributes `SrcIP`, and `time`. Detecting a Distributed Denial of Service (DDOS) attack requires the ability to compute the total number of incoming packets from each unique source IP address. In our example HiFi-based system, this counting can be done with a periodic sliding window aggregate written in CQL [2] as shown in Query 1. Here the `RANGE` indicates a computation width of r and the `SLIDE` requires results

to be reported every s time units. The actual range and slide will depend on the requirements of the user. For instance, MYSQLIDS [9] is a post-processing tool that uses static Snort data in a relational database with an interface that lets a user pick a range interval for analysis. Such a tool can be changed to run on streams where users may choose different time windows and reporting frequencies. In a large system it is likely that many concurrent users of such a tool could issue queries that compute over and report at different intervals.

Query 1 *Count packets from each unique source IP.*

```
SELECT   SrcIP, count(*)
FROM     Snort S [RANGE 'r' SLIDE 's']
GROUP BY S.SrcIP
```

2.2 Aggregates on windowed data streams

In this paper we consider aggregates that can be evaluated with constant state independent of the size of their input. Such aggregates are classified as *distributive* (e.g. `max,min,sum,count`) and *algebraic* (e.g. `avg`) aggregates by Gray [6]. Such aggregates 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 [12]) and streams (e.g., STREAM [1]).

There are two main types of windows used with streaming aggregate queries. While non-periodic windows have been studied extensively in single-site systems, periodic windows can reduce communication costs in a distributed system.

Non-periodic windows. A window is non-periodic if it has no `SLIDE` clause. For such a query the system must return an aggregate computed over the specified range whenever a client application demands it. Here successive client requests can result in computing aggregates over overlapping window intervals of the input. Arasu [1] presented general techniques to process many non-periodic aggregates in shared fashion in a single-site system. It is hard to adapt these methods to periodic windows as they consume large amounts of space that is proportional to the buffer size of the maximum window of data across all queries.

Periodic windows. A window is periodic if it includes a `SLIDE` clause. For example, in Query 1, the windows have a width (or range) defined by r and a periodicity (or slide) defined by s. Such queries can be classified 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.

In our implementation and in the rest of this paper, an aggregate operator assumes the existence of heartbeats in its input so that it can produce results even if its input rate falls off, or if there is a highly selective predicate before the

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

aggregation [7]. Golab et al. [5] present a general treatment of result production for streaming operators.

This paper focuses on aggregates that use periodic overlapping windows of streams. With non-overlapping 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.

While we focus on the harder problem of shared processing of aggregates on overlapping windows,[2] our techniques apply equally well to non-overlapping windows. The general theme of our approach toward evaluating an aggregate with an overlapping windows is to *rewrite* it so that it uses a *partial aggregate* over an appropriate non-overlapping window. Next, in Sect.3, we show how to identify non-overlapping components of an overlapping window query.

3 Identify Non-overlapping Component: Paired Windows

In this section we describe how to rewrite an aggregate query with a periodic overlapping window so that it uses a subquery with a non-overlapping window that we call a *sliced window*. We develop a new *paired window* rewrite that is superior to prior work. Paired windows are also crucial building blocks for efficient shared computation and communication, as we show in Sects. 4 and 5. We first formally define overlapping and sliced windows. We next explain paired windows and finally show how to execute aggregates with such windows.

3.1 Formal definitions

We now formally define overlapping and non-overlapping sliced windows.

Definition 1 (Overlapping windows). *An overlapping window W with range r and slide s ($r > s$) is denoted by $W[r, s]$. An aggregate over W produces a result at times t over tuples in intervals defined as a periodic function:*

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

Definition 2 (Sliced window). *A sliced window W is specified by a vector of $|W| = n$ slices and denoted by $W(s_1, \ldots, s_n)$ where the period s is the sum of all n slices. We say that each slice s_i has an edge $e_i = s_1 + \cdots + s_i$. An aggregate over W produces a result at each slice edge over tuples in the slice intervals defined for $1 \leq i \leq n$ as a periodic function:*

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

[2] While windows can be time-based or count-based we only study the former here.

3.2 Sliced Windows: Paned vs Paired

We now describe how to rewrite an aggregate query containing an overlapping window so that it uses partial aggregates with a non-overlapping sliced window. We consider two techniques, *paned windows* introduced by Li [10] and our new *paired windows* approach. Finally, we show how our paired windows always perform better than, or at least as good as, paned windows.

For ease of exposition we use an ungrouped aggregate query for our analysis in this and following sections (see Query 2 below). Our techniques, however, work for both grouped and ungrouped queries, and we show experimental results for this in our performance study in Sect. 6.

Query 2 *Count packets from all source IP addresses.*

```
SELECT   count(*)
FROM     Snort S [RANGE 'r' SLIDE 's']
```

The basic idea for efficient execution of overlapping window aggregates stems from the observation that an aggregate over a window $W[r, s]$ can be computed from partial aggregates with a sliced window $V(s_1, \ldots, s_k, \ldots, s_n)$ with period s, if and only if, $s_k + \cdots + s_n \bmod s = r$ as proved in Lemma 1 in Appendix A. The paned and paired windows with period s that can be used to rewrite the window $W[r, s]$ are defined as:

1. *Paned window:* $U(g, \ldots, g)$ where g is greatest common divisor of r and s.
2. *Paired window:* $V(s_1, s_2)$ where $s_2 = r \bmod s$ and $s_1 = s - s_2$.

An aggregate over a window $W[r, s]$ can be computed from partial aggregates over a paned or paired windows as defined above (for a proof, please see Corollaries 1 and 2 in Appendix A). While paned windows break a window of period s into s/g panes of equal size g, paired windows split a window into exactly two unequal *slices* that we call a "pair". Figure 3 shows how Query 1 with a window $W[18, 15]$ can use a paned window of 5 slices $W'(3, 3, 3, 3, 3)$ or a paired window $W''(12, 3)$.

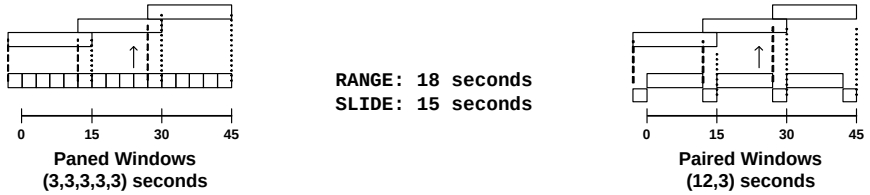


Fig. 3. Paned vs Paired Windows

We now introduce `SLICES`, a new clause analogous to the `RANGE-SLIDE` clause, to express aggregate queries with sliced windows.[3] With paned windows, Query 2 would be rewritten into Query 3 with two parts: (1) a *subquery* `Spwsub` using *partial aggregates* with a slice g (3 sec. in the example), and (2) a *superquery* that

[3] While it is possible to express a sliced window as a union of hopping windows, this syntatic sugar is more convenient.

computes the actual aggregate over the subquery's partial aggregates. We use a similar approach with paired windows where the only difference is the values used in the `SLICES` clause as shown in Query 4. In both cases we use the SQL standard `WITH` clause to express the subquery.

Query 3 *Paned window rewriting*

```
WITH Spwsub as
  (SELECT   count(*) as cnt
   FROM     Snort [SLICES '3'])
(SELECT   sum(cnt)
 FROM     Spwsub [RANGE '18' SLIDE '15'])
```

Query 4 *Paired window rewriting*

```
WITH SpairSub as
  (SELECT   count(*) as cnt
   FROM     Snort [SLICES '12 sec','3 sec'])
(SELECT sum(cnt)
 FROM   SpairSub [RANGE '18 sec' SLIDE '15 sec'])
```

3.3 Analyzing and Executing Sliced Window Aggregates

We now analyze the relative complexities of the paned and paired window approaches and then show how aggregates over arbitrary sliced windows can be efficiently processed.

As with most algorithms, the interesting tradeoffs are in space and time complexity. We measure the former in terms of the maximum aggregate state maintained at any point of time and the latter in terms of the number of aggregate operations carried out in each time interval. In particular, we count the number of times a tuple is *accumulated* in an aggregate and ignore the per-window aggregate *initialization* and *finalization* calls. Further, we assume that tuples enter the system in timestamp order (or are timestamped by the system on entry) at a data rate of λ tuples per second.

The superquery is evaluated using the Explicit Window-ID (EWID) approach proposed by Li [11]. Here, an overlapping aggregate is sent tuples augmented with an identifier (the *ewid*) encoding all windows it belongs to. It then groups these tuples by their *ewid* and aggregates them. Unlike a normal group-by, each tuple is accumulated into multiple groups (one for each window to which it belongs), using at most (r/s) aggregate state irrespective of the input data rate. The superquery has a time complexity of $r\beta/s$ where β is the data rate of the subquery. A sliced window subquery with k slices and period s uses constant state with λ operations per second and produces a stream at a rate of k/s partial aggregate tuples per second.

The key difference between the paired and paned windows is in the number of slices a window period is broken into. 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. The overall space and time costs of paned and worst-case costs of paired windows are summarized in Table 1. In the worst-case for paired windows,

$1/g < 2/s$ and in the best case (when $r \bmod s = 0$) $1/g$ is the same as $1/s$. Since the paired window never has more slices than the paned window, the paired window subquery always has a data rate no larger than that of the paned window subquery. So, paned windows are always either slower than, or at most equal to, our paired window technique.

Table 1. Complexity of paned/paired windows

Method	Time	Space
Paned window	$\lambda + (1/g)(r/s)$	$1 + r/s$
Paired window (worst-case)	$\lambda + (2/s)(r/s)$	$1 + r/s$

Executing sliced window aggregates. We now show how to execute a single aggregate query over a sliced window $W(s_1, \ldots, s_n)$ written as `[SLICES` $s_1, \ldots, s_n$`]`. The pseudocode for this operator is shown in Alg. 1. We assume that the aggregate has access to state manipulation functions such as *initialize*, *accumulate* for a new tuple, and *finaliza* to produce results. Here `init` initializes the internal state with a slice index i as 1, the internal time t representing the edge of the next slice s_i, and an aggregate state variable A. Incoming tuples y with timestamp t_y are sent to the operator by calling `next`. If t_y is smaller than the internal time t the tuple y is accumulated in A provided y is not a *heartbeat*. Otherwise, the current slice is finalized and aggregate results are emitted at slice edge t. Then, the aggregate state A is reinitialized, the slice index i is advanced, and the internal time t is set to the next slice edge.

Algorithm 1 Sliced window aggregate operator

```
proc init(W(A, i, t, s_1, ..., s_n))
  initialize(W.A); W.i ← 1; W.t ← s_1;
end
proc next(t_y, y, W(A, i, t, s_1, ..., s_n))
  if (t_y > W.t)
    then emit tuples(W.t, finalize(W.A));
         emit tuple(W.t, heartbeat);
         initialize(W.A); W.i ← W.W.i + 1 mod n; W.t ← W.t + s_i; fi
  if (y ≠ heartbeat) then W.A ← accumulate(W.A, y); fi
end
```

4 Multiple query processing

In Sect. 3 we developed the paired window rewrite to efficiently execute an overlapping window aggregate query and proved that it is always either faster than, or as fast as, the paned window approach for a single query. It turns out that the paired window approach is the optimal way of using non-overlapping aggregates to process multiple queries in a single node system.

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. For simplicity we further assume that for all i, $r_i \bmod s_i \neq 0$. For instance, the

queries in $\mathbb{Q}$ can be instances of Query 2, with different values for r and s, even where r and s are relatively prime.

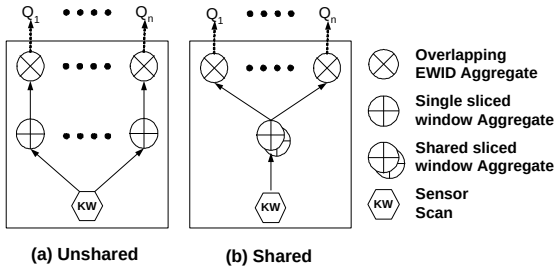


Fig. 4. Possible plans for multiple queries

4.1 Unshared versus shared aggregation

The queries in $\mathbb{Q}$ can be processed in either an unshared or shared fashion. We consider each alternative in turn:

1. **Unshared aggregation:** We rewrite each query using EWID paired windows and run them separately as shown in the query plan in Fig. 4(a). The input stream from the sensor scan is replicated to each of the n operator chains with a single sliced window aggregate feeding an overlapping EWID aggregate. Table 2 shows the space and time costs for the subqueries and superqueries. The space and time costs of each superquery is independent of the input rate λ unlike that of the n subqueries, each of which aggregate every input.

Table 2. Processing costs without sharing

Cost	Subqueries	Superqueries
Space	n	$\Sigma_i r_i/s_i$
Time	$n\lambda$	$2\Sigma_i r_i/{s_i}^2$

2. **Shared Aggregation:** We rewrite each query using EWID paired windows but avoid repeated aggregate computations of tuples in subqueries. Instead, we *compose* each of the n subqueries into a single sliced window common subquery. In Fig. 4(b) the common subquery is represented by a *shared sliced window aggregate* operator that produces a stream of partial aggregates that are replicated to the n overlapping EWID aggregate superqueries.

In the rest of this section we examine shared aggregation in detail, followed by an analysis of the relative computational costs of shared and unshared plans.

4.2 Shared aggregation

Here, we show how to compose sliced window subqueries to form a sliced window common subquery that produces partial aggregates that can be shared.

Given queries in $\mathbb{Q}$ that are rewritten using EWID paired windows, the individual paired window subqueries can be replaced by a common subquery. This

common subquery is a sliced window aggregate that is formed by composing the individual paired window subqueries. The sliced window common subquery must emit partial aggregates at every unique slice edge of each individual subquery.

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 of individual windows. With unequal periods, windows are *stretched* to the common period (*lcm*) by repeating their slice vectors.

As an example, Fig. 5 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 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^4.

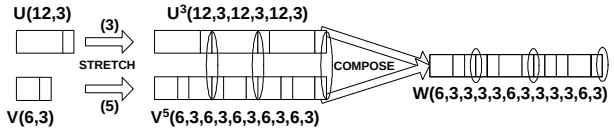


Fig. 5. Composing sliced windows

Common subquery data rate. The common subquery data rate, denoted by β, is defined as the number of partial aggregate tuples (number of unique slice edges) in each period of the common subquery. The value of β depends on the query workload. It is lowest when one sliced window subsumes all others and highest when only the final edge of all windows are shared:

$$\max_{1\leq i\leq n}(2/s_i) \leq \beta \leq \Sigma_{i=1}^{n}(2/s_i) - n + 1 \tag{1}$$

Thus, β captures the "extent" of sharing among sliced windows. In Theorem 1 we show that composing paired windows leads to a lower β than composing paned, or any other sliced, window.

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 sliced window formed by composing the paired windows of each W_i in $\mathbb{W}$. There exists no window W' formed by composing any sliced window rewriting of each W_i where $|W'| < |W|$.*

Proof: Without loss of generality, let each W_i have identical slide s (or else stretch as in Sect. 4.2). 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 (in Appendix A) 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 $|W| \leq |W'|$. □

Given β, Table 3 shows the space and time costs of the shared plan by superqueries with input rate β and the common subquery with input rate λ.

Note that a similar analysis is possible with grouped aggregation, except that the subquery data rate is not an easily analyzed constant.

Table 3. Processing costs with sharing

Cost	Subquery	Superqueries
Space	1	$\Sigma_i r_i/s_i$
Time	λ	$\beta\Sigma_i r_i/s_i$

4.3 Analysis: To share or not to share

We have explained two ways to execute multiple periodic aggregate queries. Shared aggregation (Table 3) uses less space than unshared aggregation (Table 2), as the space costs of the superqueries are identical while the number of constant size subqueries is n without sharing and only 1 with sharing. In contrast, the time complexities of both plans depend on the input rate λ, as well as the number of queries being shared and the extent of sharing. While the total cost of the superqueries without sharing is always less than that with sharing, the total cost of the unshared subqueries is always more than the cost of the shared common subquery. Analytically, we can solve for λ and say that the unshared approach costs less when inequality (3) holds.

$$n\lambda + 2\Sigma(r_i/s_i^2) < \lambda + \beta\Sigma(r_i/s_i) \tag{2}$$

$$\lambda < (\beta\Sigma_i r_i/s_i - 2\Sigma_i r_i/({s_i}^2))/(n-1) \tag{3}$$

Intuitively, sharing for a given workload is beneficial only if the input rate λ is high enough (i.e., greater than the lower bound imposed by inequality (3)). The critical factor is the "extent" of sharing in the subquery which is reflected by the common subquery data rate β. In theory, with a low input rate it may be better not to share. We consider the effects of a practical workload in Sect. 6.3.

5 Shared Communication

In Sect. 4 we examined shared and unshared plans to execute multiple aggregate queries in a streaming system. The focus in that section was on sharing computation. Here we present the best way to share communication resources while executing such queries across a hierarchy of stream processors. In particular, we show how the techniques of shared query processing apply to shared communication.

Specifically, we consider executing $\mathbb{Q}$, the set of n periodic aggregate queries from Sect. 4 in a two-level hierarchy (we consider a full hierarchy in Sect. 7). We start with the shared and unshared plans from Sect. 4. Both have three levels of operators with the overlapping aggregate on top, the non-overlapping sliced aggregate in the middle, and the sensor scan at the bottom. The network interface can be between any of these three levels resulting in three choices for aggregate location: *no push-down* where all aggregation is at the central server and none at the edges, *partial push-down* where the overlapping aggregates are

at the central server and the sliced aggregates are at the edges, and *full push-down*, where no aggregates are at the central servers and all are at the edges. This leads to six possible plans shown in Fig. 6 where a box surrounding a query plan indicates a single node in which the included plan is processed. A diamond represents a network scan operator that reads tuples from the network interface. The communication cost of each of these six plans is shown in Table 4.

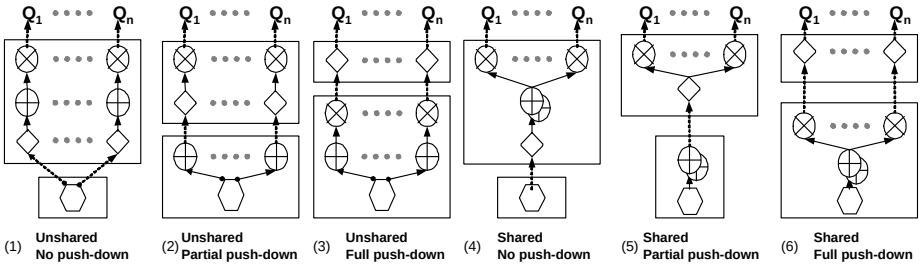


Fig. 6. Query plans for shared communication

Table 4. Communication costs for different plans

Num	Aggregation	Push-down	Bandwidth
1	Unshared	None	$n\lambda$
2	Unshared	Partial	$\Sigma_i 2/s_i$
3	Unshared	Full	$\Sigma_i 1/s_i$
4	Shared	None	λ
5	Shared	Partial	β
6	Shared	Full	$\Sigma_i 1/s_i$

Of these six plans, the two with no push down, (1) and (4) fetch raw data from lower to higher level nodes and are competitive only with low data rates, when any approach works well. Since we are interested in high data rates we do not consider these further. Since (2) is computationally identical to (3) and consumes twice as much bandwidth, we ignore (2) henceforth. With respect to communication cost, (3) and (6) are identical, and so we only consider the latter (6), as it also shares computation.

Thus, the lowest communication costs are with either shared partial push-down (5) or shared full push-down (6). With partial push down we share communication and computation resources, while with full push-down we only share computation. Shared communication pays off when the common subquery data rate is less than the total bandwidth of each fully pushed-down aggregate:

$$\beta < \Sigma_i(1/s_i) \tag{4}$$

We know from (1) that $\max_i(2/s_i) \leq \beta \leq \Sigma_i 2/s_i$. That is, the common subquery is formed by composing paired windows, each of which consume double the bandwidth of their full push-down equivalent. If the extent of sharing is high

enough, then β can be less than the upper bound in (4) above. We examine the "extent" of sharing experimentally in our performance study in the next section.

6 Performance study

In this section we report the results of a detailed performance study that investigates the benefits of two techniques: shared communication of partial aggregates in a two-level hierarchy and shared computation of aggregates in a single node. For our experiments we implemented sliced aggregation in the TelegraphCQ [8] system and deployed it on a cluster of quad 500 MHz Pentium-III nodes.

6.1 Experimental setup

We collected the logs of a Snort [14] sensor installed in the nodes of PlanetLab [13] to track incoming TCP SYN packets.[4] For our experiments we used the logs from a single node collected in a 24 hour period beginning at 4:00 am on May 1, 2005. There were 523761 tuples in this trace. In our experiments we run workloads of synthetic query sets with our implementation. Each of our workloads is a query set characterized by the parameters in Table 5.

Table 5. Query workload properties

Param.	Description	Values
q	Query	{Grouped, Ungrouped} Aggregate
n	Num. queries	$\{2, 4, 8, 16, 32\}$
r	Window range	$(1500, 2500)$ seconds
s	Window slide	$(1000, 2000)$ seconds

Our workloads have two kinds of queries - the grouped aggregate of Query 1 and the ungrouped aggregate of Query 2. All n queries in a given workload have identical slides (s) and varying ranges (r) generated uniformly from their respective intervals. In addition, all queries are overlapping - i.e., no query where r is smaller than s is generated. For each query set size, we generate 50 individual query sets. For each workload we run the queries in each of the following ways.

1. Unshared aggregate (with paired windows)
2. Shared aggregate (with a common paned window subquery)
3. Shared aggregate (with a common paired window subquery)

For each aggregate operator we measure the number of tuples it produces, as well as the total wall-clock time it consumes. From this we can compute:

1. **Communication cost** for full and partial push-down. This is in terms of the average number of tuples that have to be streamed from the lower level to upper level node in a hierarchy.
2. **Computation cost** for shared and unshared plans in a single node. This is in terms of the average total time consumed in aggregation.

[4] The inherent load on PlanetLab makes Snort drop some data.

6.2 Communication costs

We now present the results of a study of communication costs of the shared partial push-down plan (5) and the shared full push-down plan (6) from Sect. 5. Communication is shared only in the former. The results of our study are plotted in Fig. 7. Here we also consider the full aggregate pull-up strategy where raw data gets streamed to the top level node.

In the ungrouped aggregate case (shown on the left in Fig. 7), full push-down is slightly more efficient than paired partial push-down. We do not plot the cost of pull-up and paned partial push-down as they are very much more expensive (523761 and 86278 tuples respectively). For 2 queries, full push-down only needs to stream 118 tuples as opposed to 177 tuples of partial push-down. With more queries, however, the difference between the two plans drops. For instance, with 32 queries, full push-down streams 1898 tuples as opposed to 1920 tuples of partial push-down.

In the grouped aggregate case (shown on the right in Fig. 7), full push-down is cheaper than full pull-up upto 8 queries (512535 vs 523761 tuples) and cheaper than paned partial push-down upto 4 queries (257675 vs 455401 tuples). Paired partial push-down is however, far more efficient than all the other schemes, needing between 86472 tuples for 2 queries and 183910 tuples for 32 queries.

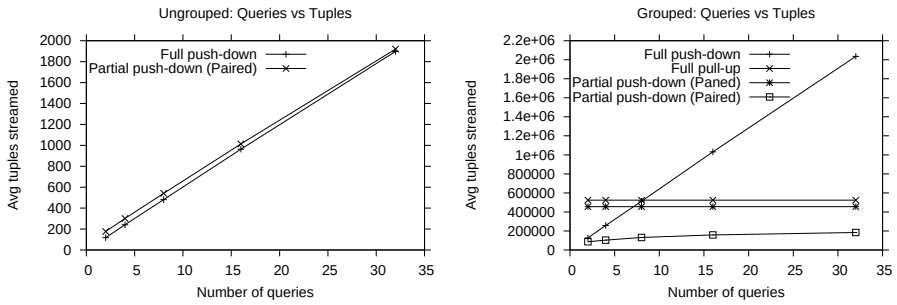


Fig. 7. Communication: Partial vs Full push down.

We now summarize the results of the study of communication costs for different query plans over a two-level hierarchy:

1. With ungrouped aggregation, partial push-down is competitive with (while slightly more expensive than) full push-down. As the number of queries increase, costs of full push-down approach those of partial push-down. In either case, however, the actual costs are very low - under 2000 tuples over a 24 hour period. Thus either approach would work well enough.
2. With grouped aggregation, partial push-down performs significantly better than the other approaches. In particular, full push-down scales very poorly with more queries.

6.3 Computation costs

Here we present the results of a study of the unshared aggregate and shared aggregate plans (from Fig. 4), each using paired window subqueries. We also

considered shared aggregation using a paned window common subquery. In addition, we measured the costs of the paired window common subquery component of the shared aggregation plan. The results of the study are plotted in Fig. 8.

In the ungrouped aggregate case (shown on the left in Fig. 8), the average computation time of all three approaches increases with more queries, as expected. The shared aggregate with paired window subqueries ("shared paired"), however, significantly outperforms the other two plans. For instance, at 2 queries the average time is 11 seconds for "shared paired" as opposed to 13 seconds for "shared paned" and 17 seconds for "unshared". At 32 queries, these differences are magnified and the average time is 39 seconds for "shared paired", 100 seconds for "shared paned" and 241 seconds for "unshared". Note that the cost of the paired common subquery remains stable from 8.9 seconds for 2 queries through 9.1 seconds for 32 queries.

In the grouped aggregate case (shown on the right in Fig. 8), the average compututation time of all approaches increases with more queries as with ungrouped aggregation (albeit at a higher rate). Here "unshared" slightly outperforms "shared paned" for all query sizes, from 53 vs 94 seconds for 2 queries through 1013 vs 1120 seconds for 32 queries. The "shared paired" approach is significantly cheaper than the other two for all queries and costs from 33 seconds for 2 queries through 505 seconds for 32 queries. The cost of the paired common subquery is stable from 18 seconds for 2 queries to 23 seconds for 32 queries.

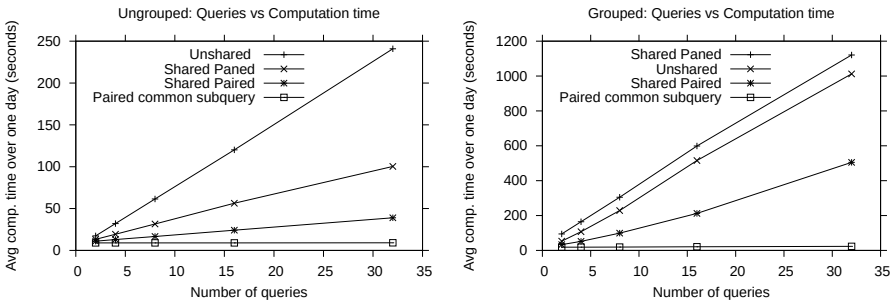


Fig. 8. Computation: Shared vs Unshared

We now summarize the results of the study of computation costs for shared aggregates with different query plans:

1. With grouped and ungrouped aggregation our "shared paired" approach significantly outperforms the "unshared" and "shared paned" approaches.
2. In our workloads, the paned common subquery almost always reduces to a `[SLICES '1 sec']` subquery showing how the paned window rewriting can make sharing very difficult.
3. In both cases, the costs of the paired-subquery component of the shared-paired aggregation remains stable with increasing numbers of queries.

7 Putting it all together

Now we apply the insights of this paper to a typical HiFi system that consists of distributed receptors on the "edge" of a network, intermediate nodes as well as a central enterprise server. The edges handle heavy loads and sharing is vital to reduce their overheads. The most important lessons learned from our experiments in Sect. 6 are:

1. Shared communication has real benefits with multiple aggregates using a partial push down strategy. These benefits *grow* with more queries.
2. Shared computation has benefits with multiple aggregates for high enough data rates. With such high rates (as in our study) the benefits of sharing outweigh the costs which also *increase* with more queries.

The second lesson in particular is striking as the overheads of sharing are normally fixed and not variable. Thus sharing can be a concern with low data rates in tiny devices where computation is expensive. The graphs in Fig. 8, however, reveal that the common paired window subquery has an almost fixed cost even with increasing queries. So it is only superquery computation (which is really unshared) that can worsen because of sharing, giving us our strategy:

1. Partial non-overlapping aggregates across time are pushed down right into the leaves (receptors).
2. Intermediate nodes, from the edges to the root, aggregate share partial aggregates across *space*.
3. The overall superqueries are only executed in the central server.

This approach has the following very desirable properties: (1) communication is shared everywhere it matters from tiny sensors to wired networks, (2) intermediate nodes execute a single shared aggregate that would not be possible with full push-down, (3) computation is unshared only at the highest level where capacity is higher, and (4) overall computation across time is only carried out at the level which issued the original query.

8 Conclusions

Widely dispersed monitoring networks generate large volumes of streaming data that can be managed through *hierarchical aggregation* - a technique to successively collect and aggregate data from distributed sources receptors through a hierarchy of data stream processors. We are building HiFi, a system to support monitoring applications over distributed data streams. In such applications there are typically a large number of users posing concurrent queries. These queries are often similar and compute the same aggregate function over different periodic overlapping windows. It is vital to share computational and communication resources by exploiting the similarities in a query workload in order to support large numbers of concurrent queries.

In this paper we showed how to share the processing of many similar aggregate queries on periodic, overlapping windows across a heterogenous hierarchy. It turns out that the obvious choices, aggregate push-down and pull-up, are both

unsuitable. While push-down reduces sharing, pull-up increases communication costs. Our solution is *partial push-down* of aggregates across a hierarchy. First, we developed the novel *paired window* rewrite (superior to prior work) to extract non-overlapping components of aggregate queries. Next, we show how to compose these non-overlapping components to share computation. Finally, we push the composed non-overlapping component down the hierarchy and pull-up the overlapping aggregates to the root. We also conducted a detailed performance study with real-world data in order to validate our technique.

Our new techniques are crucial for widely distributed monitoring networks to scale and support large numbers of concurrent queries as demanded by emerging applications.

References

[1] A. Arasu et al. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 336–347. 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*, 97–105. 1987.

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

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

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

[7] M. A. Hammad, et al. Efficient pipelined execution of sliding window queries over data streams. Tech. Rep. CSD TR#03-035, Purdue. 2003.

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

[9] R. K. Lewis. MYSQLIDS - a quick look approach to intrusion deteection systems. http://www.codeproject.com/internet/mysqlids.asp.

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

[11] J. Li, et al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. 2005.

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

[13] L. Peterson, et al. A blueprint for introducing disruptive technology into the internet. In *HotNets*. 2002.

[14] M. Roesch. Snort – lightweight intrusion detection for networks. In *USENIX LISA*. 1999.

A Proofs

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:*

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

Proof: Aggregates over $W[r,s]$ are computed over intervals of $|r|$ at times 0 mod s. We first note that:

$$r = r \bmod s + \lfloor r/s \rfloor s \tag{5}$$

(If): Suppose $s_k + \cdots + s_n$ is r mod s, from (5):

$$r = (s_k + \cdots + s_n) + \lfloor r/s \rfloor (s_1 + \cdots + s_n) \tag{6}$$

Thus an aggregate over an interval $|r|$ can be computed at times 0 mod s with partial aggregates over these $\lfloor r/s \rfloor n + n - k + 1$ contiguous slices of V:

$$\{s_k, \ldots, s_n\}, \underbrace{\{s_1, \ldots, s_n\}, \ldots, \{s_1, \ldots, s_n\}}_{\lfloor r/s \rfloor \text{ times}}$$

(Only if): Suppose W can be computed from partial aggregates over V. To aggregate all input tuples we must use a set of contiguous slices from V. Since V has period s, these slices must include an integral number (b) of complete slices for each period and a residual set of slices ($s_a, \ldots, s_n$ where $a \leq n$) from the previous period:

$$\{s_a, \ldots, s_n\}, \underbrace{\{s_1, \ldots, s_n\}, \ldots, \{s_1, \ldots, s_n\}}_{b \text{ times}}$$

The sum of these slice widths must be:

$$r = (s_a + \cdots + s_n) + b(s_1 + \cdots + s_n)$$

As $s_a + \cdots + s_n \leq s$, $b = \lfloor r/s \rfloor$ and $s_a + \cdots + s_n = r - \lfloor r/s \rfloor s$. So there exists $k = a \leq n$, $s_k + \cdots + s_n = r$ mod s. □

Corollary 1 (Paned Windows). *An aggregate over $W[r,s]$ is computable from partial aggregates over $V(g, \ldots, g)$ where g is* $\gcd(r,s)$, *$s = gn$, $r = gp$, and $n < p$.*

Proof: From number theory we know that $g = (r \bmod s, s)$. Thus, r mod $s = mg$ where $m < n$. So $\exists k, k = n - m + 1 < n$, $0 < k < n$ and $s_k + \cdots + s_n = r$ mod s. (From Lemma 1). □

Corollary 2 (Paired windows). *An aggregate over $W[r,s]$ can be computed from partial aggregates over $V(s_1, s_2)$ where $s_1 + s_2 = s$ and $s_2 = r$* mod *s.*

Proof: Set $k = n = 2$ in Lemma 1. □