Optimized Processing of Multiple Aggregate Continuous Queries

Shenoda Guirguis [1], Mohamed A. Sharaf [2], Panos K. Chrysanthis [1], Alexandros Labrinidis [1]
[1]Department of Computer Science, University of Pittsburgh
[2]School of Information Technology and Electrical Engineering, The University of Queensland
{shenoda, panos, labrinid}@cs.pitt.edu, m.sharaf@uq.edu.au

ABSTRACT

Data Streams Management Systems are designed to support monitoring applications which require the processing of hundreds of Aggregate Continuous Queries (ACQs). These ACQs typically have different time granularities, with possibly different selection predicates and group-by attributes. In order to achieve scalability in the presence of heavy workloads, in this paper, we introduce the concept of *'Weaveability'* as an indicator of the potential gains of sharing the processing of ACQs. We then propose *Weave Share*, a cost-based optimizer that exploits weaveability to optimize the shared processing of ACQs. Our experimental analysis shows that *Weave Share* outperforms the alternative sharing schemes generating up to four orders of magnitude better quality plans. Finally, we describe a practical implementation of the *Weave Share* optimizer.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Data Streams Management Systems, Aggregate Continuous Queries, Multiple Query Optimization, Weaveability, Shared Processing

1. INTRODUCTION

Data Stream Management Systems (DSMSs) were developed to be at the heart of every monitoring application, from environmental and network monitoring, to disease outbreak detection, financial market analysis and studying of cosmic phenomena (e.g., [3, 9, 1, 2, 21, 22]). DSMSs are designed to efficiently handle unbounded streams with large volumes of data and large numbers of continuous queries (i.e., exhibit scalability). Thus, optimizing the processing of multiple continuous queries is imperative for DSMSs to reach their full potential.

In all monitoring applications, hundreds of Aggregate Continuous Queries (ACQs) are typically registered [17] to monitor few input data streams. For example, a stock market monitoring application allows numerous users each to register several monitoring queries. For instance, traders interested in a certain stock might register ACQs to monitor the `average`, or `maximum`, volume trade in a certain period of time, e.g., the last 1, 8, or 24 hours. Meanwhile, decision makers might register monitoring queries for analysis purposes with coarse time granularity, e.g., the `average` volume trade in last week or month. Given the commonality of ACQs, and the high data arrival rates, optimizing the processing of ACQs is crucial for scalability.

In general, Multiple Query Optimization (MQO) is well known to be NP-hard for traditional database systems [19] as well as for DSMSs [25]. MQO techniques are typically based on heuristics that aim to share the processing of *common sub-expressions*. This raises the challenge of identifying which sub-expressions are beneficial to share, if any [27, 11]. However, the optimization of multiple ACQs goes beyond the classic identification of common sub-expressions to exploiting the window semantics and partial aggregation; this is the challenge we are addressing in this paper.

An ACQ is typically defined over a certain window of the input data stream, to bound its computations. For example, an ACQ that monitors the average volume trade of a stock index could be defined over a window of range 24 hours and slide 1 hour: every hour, the average volume trade in the past 24 hours is reported. Partial aggregation has been proposed to optimize the processing of ACQs [13, 14, 6] by minimizing the repeated processing of overlapping windows. Partial aggregation has also been utilized to share the processing of multiple similar ACQs with different windows, assuming it is always beneficial to share the partial aggregation [12].

The assumption that it is always beneficial to share is based on the premise that data arrival rate is the predominant factor in determining the sharing decision, where a high rate is always a precursor for plan sharing. It is also based on the observation that, in most practical applications, data streams do in fact exhibit a high rate. This approximation, however, considers only one facet of the problem (i.e., the characteristics of data streams) while diminishing the impact of the other facet (i.e., the characteristics of the registered queries). In this paper we argue that the properties of the installed ACQs are of equal importance in determining the sharing decision. In fact, we show that for many practical cases, sharing the partial aggregation phase of query plans could lead to a performance degradation even when data streams arrive at high rate.

To address this issue, we identify the trade offs in optimizing the shared processing of multiple ACQs and introduce the concept of *"Weaveability"* of ACQs. The *Weaveability* of a set of ACQs is an indicator of the potential gains from sharing their processing. The goal is then to design a multiple ACQs optimization technique that utilizes weaveability in the sharing decision. Exploiting weaveability, however, involves three major challenges. First, given that the optimization problem at hand is NP-hard, the technique should efficiently search the exponential search space, while not sacrificing the quality of the optimized plans. Second, the technique should also efficiently handle the addition and deletion of ACQs as time advances. Finally, it should exploit the weaveability of ACQs with minimal overhead.

In this paper, we address all the above challenges and propose *Weave Share*, a cost-based multiple ACQs optimizer, which exploits weaveability to optimize the shared processing of ACQs. *Weave Share* considers all factors that affect the cost of the shared query plan. It selectively groups ACQs into multiple execution trees to minimize the total plan cost. We experimentally evaluate and analyze the performance of *Weave Share* in terms of quality of the generated plans using all possible settings of workload characteristics. Our experimental analysis shows that *Weave Share* generates up to orders of magnitude better quality plans compared to the alternative sharing schemes. In order to handle the addition and deletion of ACQs, we propose *Incremental Weave Share* that efficiently weaves the new ACQs into the execution trees of an existing plan, as long as the quality of the query plan is not compromised, i.e., remains within specified tolerance limits. Finally, we develop and experimentally evaluate a practical implementation and several optimizations that dramatically improve the efficiency of *Weave Share* in generating high quality plans.

Contributions: In summary, we make the following contributions:

- We introduce the concept of *"Weaveability."*
- We propose *Weave Share* and *Incremental Weave Share* cost-based multi ACQs optimizers.
- We develop practical implementation and several optimizations of the *Weave Share* optimizer.
- We experimentally evaluate our proposed techniques using simulation.

Road map: We summarize related work in Section 2 and describe the streams aggregation model in Section 3. We formalize the problem and introduce *Weaveability* in Section 4. We present *Weave Share* and its online version in Section 5. The evaluation platform and the quality of weave share plans are discussed in Sections 6 and 7, respectively. We present implementation optimizations in Section 8 and conclude in Section 9.

2. RELATED WORK

There is a rich literature on multiple query optimization (MQO) in traditional databases [20, 18, 16, 10, 23] as well as in data streams [23, 1, 24, 15]. Finding the optimal query plan in traditional databases is an NP-Hard problem [19]. Hence, cost-based heuristic approaches have been investigated [18, 8]. This is also the case in data streams. In the context of multiple ACQs optimization, two orthogonal approaches were proposed: *scheduling* and *partial aggregation*. In [7], a window-aware scheduling scheme was proposed to synchronize the re-execution times of similar ACQs to execute common parts only once.

Partial aggregation is the underlying principle for an efficient implementation of the aggregate continuous operator [13, 12]. The Panes [13] scheme splits the slide into equal sized fragments, to be processed using the partial aggregation operator. Paired windows [12] improves Panes by splitting the slide into exactly two fragments, minimizing the processing needed at the final-aggregation level. We assume the paired window scheme in our model, as discussed in the next section.

The work in [25, 26, 17] proposed a shared processing of multiple ACQs with different group-by attributes. The proposed scheme maintains fine-granularity ACQs called phantoms, or intermediate aggregations. The computation of these intermediate aggregations can then be shared among the different ACQs. While in [25, 26] the proposed techniques were designed for the architecture of Gigascope, a special purpose stream processing engine, they were generalized in [17]. In principle, the general idea of grouping ACQs in a hierarchy of intermediate ACQs is conceptually similar to the selective sharing technique underlying our proposed *Weave Share*. However, while the work in [17] focuses on optimizing the processing of multiple ACQs that have different *group-by attributes*, our work focuses on optimizing multiple ACQs that have different *window specifications*. These are two orthogonal problems.

3. STREAMS AGGREGATION

In this background section, we discuss window semantics of ACQs and partial aggregation, as per the current state of the art.

3.1 ACQ Semantics

An ACQ is defined over a window, which is specified in terms of two intervals: *range (r)* and *slide (s)*. For example, an ACQ may compute the average stock price over the last hour (i.e., $r = 1$ hr) and update it every 30 minutes (i.e., $s = 30$ min). The range and slide intervals could be defined either as number of tuples or as a time interval. Here, we consider the more general time-based definition for both the range and slide; our contributions are applicable to the tuple-based definition. Producing a new window requires processing each tuple within the range. The slide interval defines how the window boundaries move over the input stream. For instance, when slide is less than range (sliding window), consecutive windows overlap and a single tuple will belong to more than one window instance. This is illustrated in the example below.

EXAMPLE 1. *Consider a stock monitoring application where the user is interested in the average trade volume in the past hour, and would like to be updated every ten minutes. The user then registers an ACQ with $r = 1$ hr and $s = 10$ min. Thus, a window boundary is reached every 10 min and an aggregation is performed over the tuples within the last hour. Hence, each input tuple is aggregated in six consecutive windows (=1 hr/10 min).*

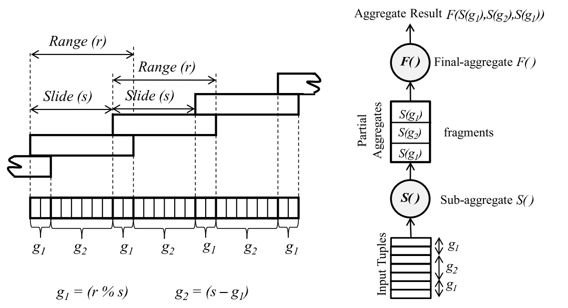
In a straightforward implementation of ACQs, input tuples are buffered and once a window boundary is reached, the aggregate function is evaluated using the tuples that are within the range boundaries. Then as the boundaries are shifted, all tuples that fall outside the new boundaries are expired and discarded.

3.2 Partial Aggregation

Clearly the straight forward implementation of ACQs is inefficient. *Partial-aggregation* [13, 12] has been proposed to efficiently process ACQs. Under partial-aggregation, the final aggregate value is assembled from a set of partial aggregate values. For example, under partial aggregation an aggregate `COUNT(*)` is computed using (1) a `COUNT(*)` on each partition and (2) a `SUM(*)` over the partial counts. Clearly, partial aggregation is applicable over all distributive and algebraic aggregate functions that are widely used in database systems, such as: `MAX`, `COUNT`, `SUM`, etc.

In general, for a dataset G of disjoint fragments $g_1, g_2, ..., g_n$, an aggregate function A over G can be computed from a *sub-aggregate* function $\mathcal{S}$ over each dataset g_i and a *final-aggregate* function $\mathcal{F}$ over the partial aggregates. Formally,

$$A(G) = \mathcal{F}(\{\mathcal{S}(g_i) | 1 \leq i \leq n\}).$$



(a) Fragments (b) Plan

Figure 1: Paired Window Technique

Partial aggregation reduces the overall query processing cost by processing each input tuple only once by the sub-aggregate operator to produce partial aggregates. As the window slides, only partial aggregates are buffered and processed to generate new results. Clearly, smaller number of partial aggregates means fewer final aggregate operations. The *paired window* technique [12] (Figure 1(a)) partitions each slide into at most two *fragments* g_1 and g_2 (i.e., a pair). Hence, producing a final aggregate requires at most $\lceil 2 \times \frac{r}{s} \rceil$ operations, where $\frac{r}{s}$ is the number of slides per window and 2 is the maximum number of fragments per slide.

The bottom part of Figure 1(a) shows the set of input tuples, while the top part shows different overlapping window instances. Each *slide* is paired into exactly two *fragments* of length: g_1 and g_2, where $g_1 = r\%s$ and $g_2 = s - g_1$. Given this partitioning, the range consists of a sequence of $g_1, g_2, ..., g_1$ fragments, where the length of a paired window equals $g_1 + g_2 = s$. Note that if r is a multiple of s, then only one fragment is produced per slide. Figure 1(b) illustrates the query plan of the ACQ in Figure 1(a). The end of each fragment g_i represents an *edge*, where the tuples in g_i are assembled into a partial aggregate.

4. WEAVE SHARING OF MULTIPLE ACQs

In this section, we introduce the concept of *Weaveability* (Section 4.3), after first illustrating the trade-off involved in the shared processing of ACQs. We also discuss the challenges of optimizing the shared processing of multiple ACQs (Section 4.4).

4.1 Partial Aggregation Sharing Trade-off

The sharing of the partial aggregation among a set of n ACQs $\{q_1, q_2, ..., q_n\}$ with slides $\{s_1, s_2, ..., s_n\}$, respectively, involves three steps:

1. Multiple slides are integrated into a new *composite slide* (CS) of length equal to the least common multiplier of the individual slides (i.e., $CS = lcm(s_1, ..., s_n)$),
2. Each slide s_i is then stretched into a new slide s'_i of length CS, where the edges (i.e., end of each fragment) in each slide s_i are then copied and repeated to the length of s'_i (=repeated $\frac{s'_i}{s_i}$ times), and
3. The fragments in the composite slide are created by overlaying each edge from each individual slide s'_i onto the new composite slide CS, unless that edge already exists in CS (i.e., common edge).

While sharing the sub-aggregation operator reduces the cost at the sub-aggregation level, it might increase the processing needed at the final-aggregate level. To illustrate that trade-off, consider the following example.

EXAMPLE 2. *Consider two ACQs q_a and q_b with ranges $r_a = 12$ and $r_b = 10$, and slides $s_a = 9$ and $s_b = 6$ seconds, respectively. Hence,the fragments in q_a's slide are of length $g_{a,1} = 3$ and $g_{a,2} = 6$ and for q_b, the fragments are $g_{b,1} = 4$ and $g_{b,2} = 2$.*

If q_a and q_b are processed independently, their sub-aggregation operators will produce 2 fragments every 9 and 6 sec, respectively. That is, an *edge rate* (i.e., number of fragments generated per sec) of $E_a = 0.22$ and $E_b = 0.33$ edges per sec. Thus, the total final-aggregation operations performed per sec is 0.55.

Meanwhile, if q_a and q_b share their partial aggregation, then s_a and s_b are integrated into *composite slide* $CS_{a,b} = lcm(s_a, s_b) = 18$ and the union of edges in $CS_{a,b}$ will appear at times (3, 4, 6, 9, 10, 12, 16, 18). Hence, each of q_a and q_b would examine a combined edge rate of $E_{a,b} = 0.44$, resulting in more final-aggregation operations (0.88 per sec). This simple example clearly shows the presence of a trade-off in the shared processing of multiple ACQs.

We denote those two general methods above as *No Share* and *Shared*. The latter (i.e., *Shared*) is the core principle underlying the *Shared Time Slices (STC)* scheme [12]. Given that clear trade-off between these two extremes of sharing, our goal is to design a selective sharing scheme that groups ACQs into $1 \le m \le n$ groups, that is m execution trees, where the ACQs in each group are shared (as formalized in the next sections).

4.2 Formalization

Given a set of ACQs $Q = q_1, q_2, ..., q_n$ where r_j and s_j are the range and slide of each ACQ q_j, and given a grouping of the ACQs into m execution trees $t_1, t_2, ..., t_m$ where all ACQs of a tree t_i are shared, the cost, in terms of total number of aggregate operations per second, of each tree is computed by:

$$C_{t_i} = \lambda + E_i \Omega_i \quad (1)$$

where λ is the data input rate and for a tree t_i, E_i is the edge rate (i.e., number of fragments generated per second) and Ω_i denotes the total number of final-aggregation operations performed on each fragment. Hence, for a set of shared ACQs $SQ = q_1, q_2, ..., q_k$ in a tree t_i, the total number of final aggregations is computed as:

$$\Omega_i = \sum_{j=1}^{k} \frac{r_j}{s_j} \quad (2)$$

We refer to Ω_i as the *tree overlap factor* or *overlap factor* for short.

Thus the total cost of the query plan is simply the sum of the cost of the individual trees. Specifically, if the query plan contains m trees, then the total cost of the query plan is computed as:

$$C_{\text{m-trees}} = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \quad (3)$$

Note that the first term of Eq. 3 is the cost at the sub-aggregation level, whereas the second term is the cost at the final-aggregation level. The goal of our proposed *Weave Share* is to group the ACQs into a set of trees in a way that minimizes Eq. 3. That is, to strike a balance between the two components of the cost function. In particular, our objective is to find the most beneficial number of trees (i.e., m) as well as the best assignment of ACQs to each tree in order to provide the lowest execution time.

4.3 Weaveability

The affinity of ACQs, i.e., their similarity, is an important factor that determines whether it is beneficial to share two ACQs or not.

We refer to this affinity as the *weaveability* of ACQs. Specifically, given the paired-window processing scheme, two ACQs are said to be *perfectly weaveable* if the edges of both ACQs are identical. That is, when the two ACQs are shared, the edge rate does not increase for either of the ACQs. If the ACQs are not perfectly weaveable, the more *common* edges between the ACQs in their composite slide, the less the increase in edge rate for the ACQs when shared, hence the more *weaveable* they are. Thus, we define the degree of weaveability as follows.

DEFINITION 1. *Given two ACQs q_a and q_b with slides s_a and s_b, respectively, the degree of Weaveability of q_a and q_b ($\mathcal{WV}_{a,b}$) is the ratio of the number of common edges M_c in the composite slide $CS_{a,b} = lcm(s_a, s_b)$, to the total number of edges ($M_{a,b}$) in $CS_{a,b}$. Specifically,*

$$\mathcal{WV}_{a,b} = \frac{M_c}{M_{a,b}} \tag{4}$$

Note that the definition of weaveability is recursively applicable to two groups of shared ACQs, i.e., execution trees.

Sharing weaveable ACQs has the desirable property of minimizing the increase in final-aggregation cost since those weaved ACQs would have many common edges, which keeps the increase in edge rate of the final shared plan to a minimum. For example, for the two ACQs q_a and q_b (Example 2), with the set of edges of the composite slide (3, 4, 6, 9, 10, 12, 16, 18), the common edges are (12, 18). Thus, the weaveability $\mathcal{WV}_{a,b} = \frac{2}{8} = 0.25$, which is a relatively weak weaveability, and that explains why their shared tree encounter a high increase in the edge rate.

4.4 Challenges of Grouping Multiple ACQs

Grouping ACQs to multiple trees involves three major challenges. Namely: 1) designing a technique that effectively prunes the combinatorial search space, 2) handling the dynamic addition and deletion of ACQs over time, and 3) efficiently computing the weaveability with minimal overhead.

Towards the first challenge, grouping ACQs could be seen as first determining the optimal number of execution trees and then assigning ACQs to the trees. Thus, we have initially considered mapping our ACQ sharing problem to the generalized task assignment problem which is known to be NP-Hard [5]: the input is a set of heterogeneous machines and a set of tasks, where each task has a certain cost when processed on a certain machine. The output is an assignment of tasks to machines that minimizes the total cost.

This mapping, however, assumes the knowledge of number of machines (i.e., trees), which is not the case. Furthermore, even if we assume the knowledge of the optimal number of trees to use, the increase in processing cost when adding an ACQ to a tree is not constant as it depends on which other ACQs have already been assigned to that tree. This is simply true because the cost function in Eq. 3 involves the edge rate term, which depends on which ACQs are shared and the degree of weaveability of those ACQs.

Thus, we can not directly use any of the classical algorithms for solving the task assignment problem (e.g., Dynamic Programming) to solve our ACQ sharing problem. This is mainly because an optimal solution for a sub-problem is not necessarily a part of the optimal solution of the whole problem. In other words, there is no optimal substructure property.

Given the problem complexity discussed above, we have explored a suite of alternative algorithms towards the efficient sharing of ACQs. In this paper, we present *Weave Share*, an efficient heuristic that fully considers all cost factors in generating shared plans (Section 5).

```
1: Input: A set of $n$ queries
2: Output: Weaved query plan $P$ that consists of $m$ execution trees
3: begin
4: $P \leftarrow$ Create an execution tree for each ACQ
5: $l \leftarrow n$ {current number of trees}
6: $(max\text{-}reduction, t_1, t_2) \leftarrow (0, -, -)$ {current tree-pair to merge}
7: repeat
8:     for $i = 0$ to $l - 1$ do
9:         for $j = i + 1$ to $l$ do
10:            $temp \leftarrow$ cost-reduction-if-merging($t_i, t_j$)
11:            if $temp > max\text{-}reduction$ then
12:                $(max - reduction, t_1, t_2) \leftarrow (temp, t_i, t_j)$
13:            end if
14:        end for
15:    end for
16:    if $max\text{-}reduction > 0$ then
17:        merge($t_1$,$t_2$)
18:        $l \leftarrow l - 1$
19:    end if
20: until No merge is done
21: Return $P$
22: end
```

Figure 2: The *Weave Share* Algorithm

The second challenge is the need for an online version of the algorithm that handles the addition and deletion of ACQs as time advances. To handle this challenge, we propose *Incremental Weave Share*, the online version of *Weave Share* that avoids the reconstruction of the query plan every time an ACQ is added or deleted. Both *Weave Share* and its online version are discussed in the following section (Section 5).

The third challenge (i.e., computing weaveability) stems from the complexity of counting the number of common edges between two different trees. This is because when merging two trees, there is no closed-form formula that determines the common edges. Specifically, this problem maps to small sieve theory problem which is a hard problem, and whose current solutions mostly deal with approximations and there is no closed formula to solve it [4]. Yet, the degree of weaveability directly determines the amount of increase in total processing cost (if any) when merging. To efficiently consider the weaveability while generating the shared plan, we propose several optimizations for the process of counting the number of common edges (Section 8).

5. THE WEAVE SHARE OPTIMIZER

5.1 Weave Share Algorithm

Our proposed *Weave Share* exploits weaveability to reap the benefits of cost reduction provided by sharing partial aggregation, while minimizing the increase in cost incurred at the final aggregation. Basically, *Weave Share* tries to group ACQs in multiple execution trees, where each tree contains only ACQs that *weave* best together.

To achieve our goal, *Weave Share* takes a global view of the execution plan as well as the objective function to minimize (i.e., Eq. 3). In particular, it simultaneously considers both of the cost components (i.e., partial- and final-aggregation) to group ACQs in multiple trees with minimum execution cost.

Weave Share (pseudo-code in Figure 2) takes as an input a set of ACQs $q_1, q_2, ..., q_n$ and produces a set of m shared trees where each tree contains one or more ACQs. Initially, the number of trees is equal to the number of individual queries, $m = n$ and each ACQ forms a separate tree, which is equivalent to the case of no sharing.

Weave Share advances towards sharing one step at a time in a greedy manner, where in each iteration two weaveable trees are

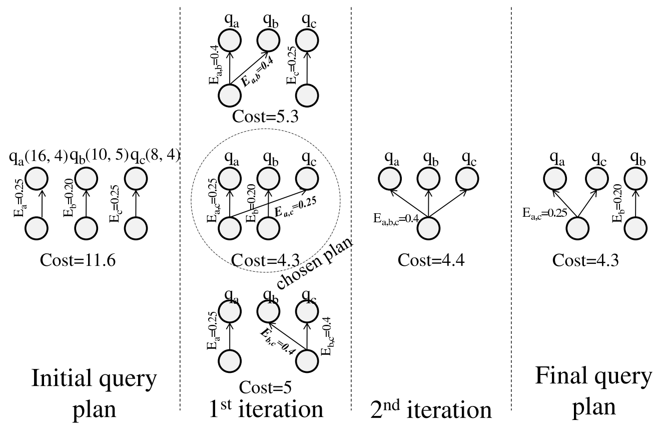


Figure 3: Example 3 - Iterations of *Weave Share*.

merged, reducing number of trees by one, until no more merging is beneficial. In particular, at each iteration, given a set T of l trees: $T = t_1, t_2, ..., t_l$ $(l \leq n)$, *Weave Share* estimates the benefits of merging all possible pairs of trees in T and merges the pair of trees that yields the maximum reduction in total cost.

Given Eq. 1, it is expected that for a pair of trees (t_x and t_y) to qualify for merging, they must satisfy either one or both of the following properties:

1. High degree of weaveability. The higher the degree of weaveability of the merged trees, the less the increase in the combined edge rate $E_{x,y}$ and the less the overall merged tree cost.

2. Low total overlap factor ($\Omega_{x,y} = \Omega_x + \Omega_y$), which is the total number of final-aggregation operations performed on each fragment in the new tree. Hence, the less the number of window instances, the less the number of final-aggregate operations performed on each edge (i.e., fragment).

The benefit (i.e., cost reduction) from merging t_x and t_y is:

$$\Delta(C_{x,y}) = \lambda + E_x\Omega_x + E_y\Omega_y - E_{x,y}\Omega_{x,y} \qquad (5)$$

Note the term λ in Eq. 5 above denotes the savings at the sub-aggregation level. That is, each tuple is processed once instead of twice. The rest of the terms in the equation represents the savings in the final aggregation level.

Clearly, any two trees that exhibit the two properties above are good candidates for merging as they allow us to exploit the sharing of partial-aggregation while at the same time minimize the increase in final-aggregation. These are the main optimization criteria for *Weave Share*. We demonstrate how *Weave Share* iterations work using Example 3 below.

EXAMPLE 3. *Consider three queries q_a, q_b and q_c with sliding window specifications as follows: ranges 16, 10 and 8 and slides 4, 5 and 4, respectively. Additionally, consider an input rate $\lambda = 1.2$ tuples per second.*

Figure 3 shows the sequence of iterations performed by *Weave Share* as well as the resulting query plan. Figure 3 shows that initially, the number of trees is three, with no sharing. This results in a total cost of 11.6 based on Eq. 3 as shown in the figure (the calculations details are omitted for brevity). Next, the algorithm enters the main loop where it tries to merge the pair of trees that would reduce the cost the most.

In the first iteration, there are three possible pair-wise merges $< q_a, q_b >$, $< q_a, q_c >$, or $< q_b, q_c >$. Merging the pair $< q_a, q_c >$ leads to the maximal reduction in cost, reducing it to 4.3 aggregations per second according to Eq. 5. Thus, the algorithm merges them together into tree $t_{a,c}$ and proceeds to the next iteration.

In the second iteration, the only possibility is to merge $t_{a,c}$ with q_b. This, however, would lead to an increase in the cost to 4.4 aggregations per second. Since there is no room for improvement, *Weave Share* terminates the loop and returns the query plan it constructed: $t_{a,c}$ and q_b, where q_a and q_c are shared in $t_{a,c}$ and q_b is executed independently. Note that q_a and q_c *weave* well together, in the sense that all the edges of q_a exist in edges of q_c (i.e., common). This is due to the fact that their slides are equal. This results in no increase in the edge rate when they are merged and in turn, minimizes the overall execution cost.

5.2 Incremental Weave Share

In the previous section, we described the basic (offline) *Weave Share* algorithm, which constructs a query plan from scratch. In this section, we consider the online case where newly submitted ACQs are weaved into the current plan and the case of re-weaving existing trees after the deletion of some ACQs.

5.2.1 Adding New ACQs

Reconstructing the *Weave Share* query plan from scratch is one possible solution to handle the submission of a new set of ACQs into the system. In that solution, given an already existing set of ACQs Q in a *Weave Share* plan P and a set of new ACQs Q', *Weave Share* is invoked to generate a new weave share plan P' which includes the ACQs $Q \cup Q'$. This solution, however, has two drawbacks: 1) it incurs a large overhead since the algorithm is reinvoked to run from scratch whenever new ACQs are added, and 2) it might often lead to an unnecessary reconstruction since, in many cases, the new plan P' can be directly achieved from the current plan P.

To address the above drawbacks, we develop *Incremental Weave Share* which takes a more *lazy* approach for maintaining the weaved plan. This involves the following two steps:

1. Immediately incorporating new ACQs into the existing plan.

2. Reconstruct the query plan from scratch *only when needed*.

In this incremental version of *Weave Share*, a new tree t_{new} is created for each new ACQ q_{new} that is added to the system and *Weave Share* is invoked to merge t_{new} with the trees in the current plan P to generate a new incremental plan P''. Thus, among the existing trees, t_{new} will be merged with the one tree with which it weaves the best. The newly merged tree might be further merged with other trees in the plan if this is beneficial. This process continues until no further improvements are attainable.

The cost of the incremental plan P'' might, however, be worse than the plan P' which would be generated by the offline *Weave Share*. In order to detect the magnitude of that degradation, *Incremental Weave Share* maintains the *performance slope* of the *plan-cost curve*. This curve is basically a plot of the offline-generated plan cost vs. the number of ACQs. The points on the curve are obtained when a plan P' is generated from scratch.

As new ACQs are submitted to the system, the cost of P'' is compared with the extrapolated cost using the performance slope. If the difference percentage is more than a certain *deviation tolerance* threshold, which is a system parameter, a reconstruction phase is triggered and performed asynchronously. Specifically, for a *de-*

viation tolerance of ϵ, a reconstruction is triggered iff:

$$\frac{cost(P'')}{\textit{extrapolated cost using performance slope}} - 1 > \epsilon \qquad (6)$$

As such, the deviation tolerance value acts as a *knob* to control the reconstruction behavior. For instance, setting the tolerance to zero, resembles reconstructing the weaved plan whenever a new ACQ is added, whereas setting the tolerance to ∞ is equivalent to the case where no reconstruction is ever performed.

5.2.2 Deleting ACQs

We handle the deletion of existing ACQs similarly to the addition of new ACQs. Specifically, deleted ACQs are first removed from their respective tree, then each of those trees is then examined against all the other trees in the weaved plan for beneficial merging, i.e., if the merge would result in a reduction in the total cost of the updated plan. This process is repeated until no more improvements are attainable. Similarly to adding ACQs, given the performance slope and a tolerance factor, a reconstruction phase may be triggered depending on the degradation from the extrapolated cost.

5.3 Varying Predicates and Group-by

Weave Share can easily handle the case when different ACQs have different pre-aggregation filters (i.e., selection operators). For example, one query might monitor the average-volume of stock-trades that are higher than \$100, while another monitors the same for trades that are higher than \$500. To share the execution of such ACQs, we adopt the *Shared Data Shards* (SDS) technique [12].

Further, when different ACQs have different group-by attributes, *Weave Share* can utilize the techniques in [17, 26]. Specifically, each sub-aggregation operator can utilize a hash table based on the values of the union of all group-by attributes. When a fragment is due, proper hash table entries are combined together to form the fragment of each set of queries with identical group by attributes.

6. EXPERIMENTAL PLATFORM

We built a simulation platform in C++ to evaluate the quality of *Weave Share* plans. We validated our simulation model by reproducing same results trends as in [12] and running Exhaustive Search to find the optimal plan for small cases that we solved by hand[1]. Below, we list the different algorithms we compared with and describe the generated workload characteristics, experiments parameters (summarized in Table 1) and the performance metrics.

Algorithms: In addition to *Weave Share* and *Incremental Weave Share*, we implemented *Random*, *Shared* and *No Share* (base-line). We also implemented an obvious simplified version of the *Weave Share* algorithm, which we call *Insert-then-Weave*. *Insert-then-Weave* is a greedy heuristic that inserts ACQs, one at a time in an arbitrary order, to the tree that it weaves best with. After this phase, a weaving phase of merging the created trees (similar to *Incremental Weave Share*) follows.

Finally, we also adapted *Local Search* (LS), which is is a sub-optimal state space search algorithm. *LS* starts from an arbitrary initial state, i.e., grouping of ACQs. We used both *Random* and *No Share* to generate the initial states. In each iteration, *LS* moves towards the optimal solution by moving a single ACQ from one tree

[1]Other experiments with Exhaustive Search showed that *Weave Share* generates mostly optimal plans; for input rates/number of ACQs of 200/5, 300/10 and 400/15, *Weave Share* generated the optimal plan. In only one case, *Weave Share* generated the optimal number of trees but 3% more costly, due to different grouping of ACQs (*Shared* plan was 32% more costly in this case).

Table 1: Simulation Parameters

Parameter	Values
Slide Length (s)	[1 – 100000] using Zipf distribution
Slide Skewness	[0.0 – 3.3] (skewed to large-slide)
Max. Overlap Factor (Ω_{max})	[50 - 2000]
Overlap Factor	[1 – Ω_{max}]
Number of ACQs	[50 – 2000]
Input Arrival Rate	[0.5 – 1,000,000] tuples/sec.

to another. In our experiments, we bound the number of iterations of *LS*, for each workload instance, to two, five or ten times the the number of iterations that *Weave Share* needed for that instance. Among these different options, *LS-NS-10x* (which starts from *No Share* initial state and uses ten times iteration's bound) was the best and we report its results.

ACQs: We generated ACQs with different specifications. The slide length (s) was drawn from a Zipf distribution over a discrete range. The discrete range depicts the real-world case of pre-specified (i.e., template) window specifications. The skewness of the Zipf distribution reflects the popularity of certain slide lengths. The range (r) for each ACQ is relative to its slide. That is, $r_i = \omega_i \times s_i$, where ω_i is the overlap factor. The overlap factor is uniformly distributed between 1.0 and Ω_{max}, which is a simulation parameter. In each experiment, we also changed the the number of ACQs and the input rate. The input rate values are chosen to cover a wide variety of different monitoring applications, ranging from phenomena monitoring (few tuples, or less per second) to high speed network monitoring (1M tuples/sec).

Dataset: We chose to use synthetic workload, which allows us to control the system parameters, in order to conduct detailed sensitivity analysis and gain better insight into the behavior of *Weave Share* by setting the parameters to cover all possible real scenarios.

Performance Metrics: We measured the quality of plans in terms of the cost of the plans as the number of aggregate operations per second (which also indicates the throughput). We chose this metric because it provides an accurate and fair measure of the performance, regardless of the platform used to conduct the experiments.

7. QUALITY OF WEAVE SHARE PLANS

7.1 Weave Share Performance

In this section we present our evaluation results, comparing *Weave Share* to other alternatives under different workload parameters.

7.1.1 Number of ACQs (Fig. 4 to 7)

Figures 4 and 5 show the cost of the *Weave Share* plan as the number of ACQs increases from 50 to 1000, for low (50 tuples/sec) and medium (300 tuple/sec) input rates, respectively. In both plots, the maximum overlap factor (Ω_{max}) is set to 50, and the slide skewness is 0.6. *Weave Share* always outperforms the best of all other algorithms. For instance, for 1000 ACQs, *Weave Share* outperforms *Insert-then-Weave* and *Shared* by three and four orders of magnitude, at low and medium input rates, respectively.

Note that *No Share* and *Random* generate the most expensive plans in both cases. *LS* did not find better plans than those generated by *Weave Share*, while incurring a very high overhead. Specifically, *LS-NS 10x* took thousand times the time needed by *Weave Share*. The reason is that an iteration of *LS* moves a single ACQ from a tree to another, while an iteration of *Weave Share* merges two trees, i.e., moves a group of ACQs at once. Thus, *Weave Share* reaches a reasonable sub-optimal solution much faster than *LS*.

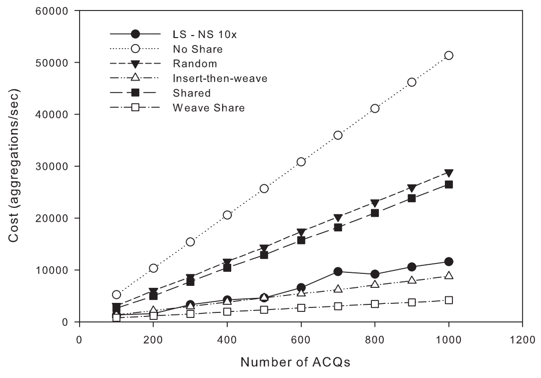


Figure 4: Impact of number of ACQs on cost for different policies under low input rate (50 tuples/sec)

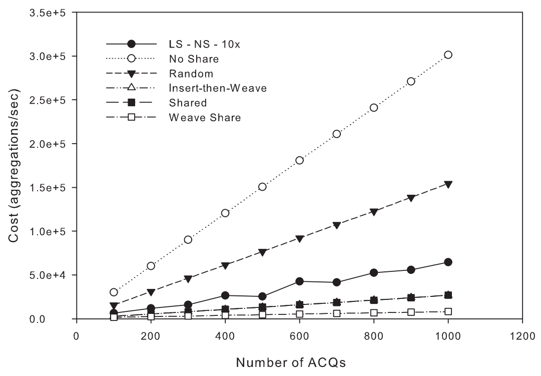


Figure 5: Impact of number of ACQs on cost for different policies under medium input rate (300 tuples/sec)

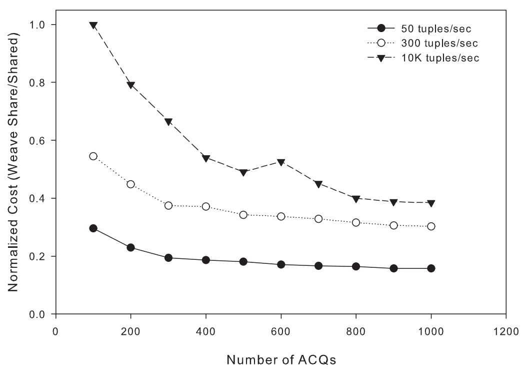


Figure 6: Impact of number ACQs on normalized cost under low, medium and high input rates

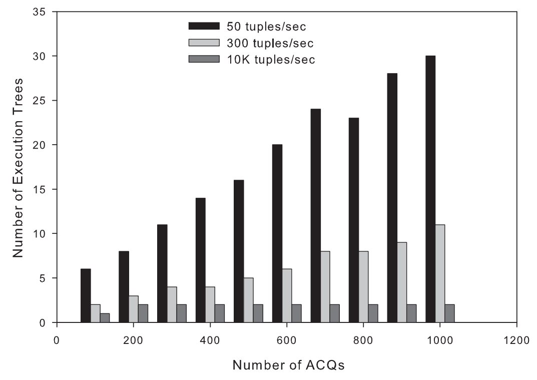


Figure 7: Number of Execution Trees for different number of ACQs under low, medium and high input rates

While *LS* and *Insert-then-Weave* perform better than *Shared* at low input rate (50 tuples/sec), *Shared* outperforms both at medium input rate (300 tuples/sec). For high input rate (10K tuples/sec), omitted for brevity, the relative performance of different algorithms was similar to the medium input rate case.

In Figure 6, we study the performance of *Weave Share* compared to *Shared*, for the low, medium and high input rates. Specifically, we plot the normalized cost of *Weave Share* plan to the cost of *Shared*, for the three input rates. The figure shows that as the number of ACQs increases, the gain provided by *Weave Share* increases. It also shows that even for high input rate (10K tuples/sec), as the number of ACQs increases, *Weave Share* outperforms *Shared*. For instance, at 1000 ACQs, for input rate of 10K tuples/sec, *Weave Share* reduces the cost by 62%.

The improvement of *Weave Share* over the best of other algorithms increases with the number of ACQs because the more ACQs, the more chances for *Weave Share* to selectively share ACQs that weave well together. Thus, limiting the increment in edge rate (E) and overlap factor (Ω) per merged tree, while still benefiting from sharing the partial aggregation. Finally, we notice in Figure 6 that for high input rate of 10K tuples/second and few number of ACQs, *Weave Share* performs identical to *Shared*, i.e., generates one tree. For such settings, the overhead at the partial aggregation level dominates that at the final-aggregation level.

Finally, Figure 7 shows the number of execution trees that were generated by *Weave Share* for the same settings as in Figure 6. As expected, the number of trees increases as the number of ACQs increases, while it decreases as the input rate increases. It also shows that for high input rate of 10K tuples/sec, *Weave Share* still generates more than one tree for more than 100 ACQs. This confirms our observation that the properties of the installed ACQs are as important as the input rate in determining the sharing decision.

7.1.2 Input Rate (Fig. 8)

In this experiment we study the sensitivity of *Weave Share* to the input rate. We report the normalized cost of *Weave Share* to that of *Shared* (here and in all the experiments hereafter) because *Shared* was the best alternative (in each experiment). We plot the normalized cost for different values of number of ACQs in Figure 8. The results in this plot are for a workload with Ω_{max} of 50 and slide skewness of 0.6.

Similar to the previous experiment, as the input rate increases, the performance of *Weave Share* approaches that of *Shared*. For instance, for 250 ACQs, the gain of *Weave Share* starts at 80% at input rate of 50 tuples/sec, and reaches 24% and 6% at input rates of 2K and 3K tuples/sec, respectively. For high input rate (10K tuples/sec), *Weave Share* still generates plans that are 12% and 24% less costly for 1000 and 2000 ACQs, respectively.

Weave Share also outperforms *Random*, *Insert-then-Weave*, *No Share* and *Local Search* by up to orders of magnitude. For instance, at input rate of 10K tuples/sec, *Weave Share* generates a plan that is more than 100 times less costly than the best of other plans.

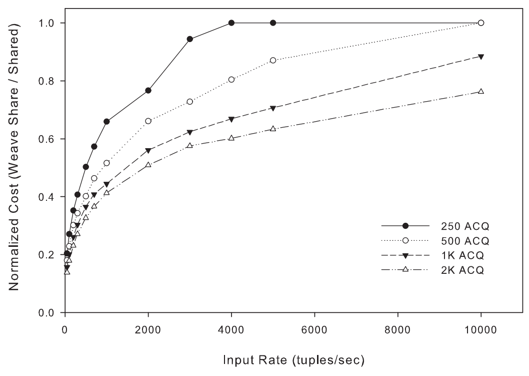


Figure 8: Impact of Input Rate on normalized cost for different number of ACQs

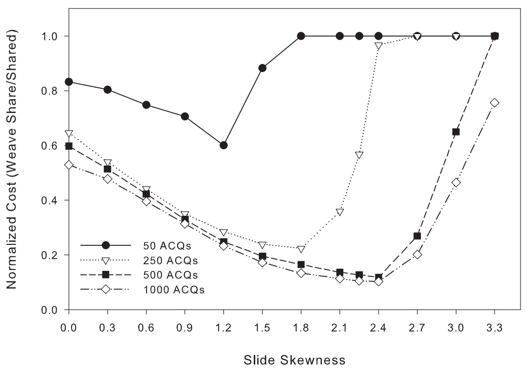


Figure 10: Impact of Slide Skewness on normalized cost for different number of ACQs

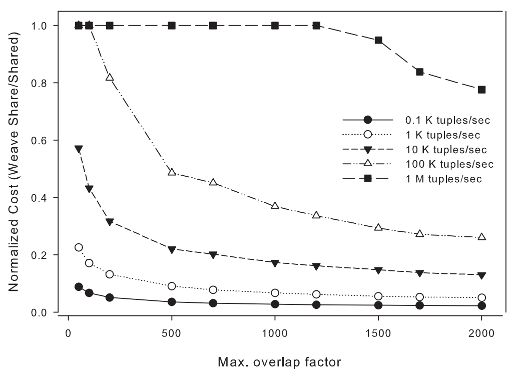


Figure 9: Impact of Maximum Overlap Factor on normalized cost for different input rates

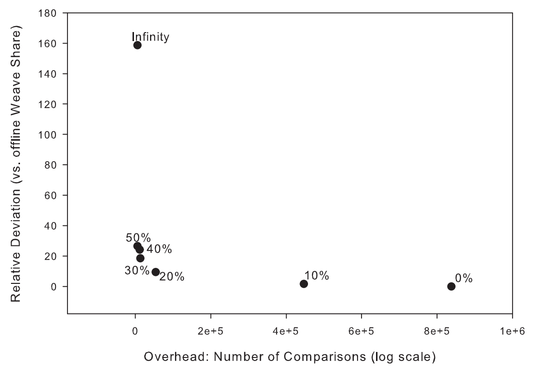


Figure 11: *Incremental Weave Share*: Deviation vs Overhead for different tolerance factor (ϵ).

7.1.3 *Maximum Overlap Factor (Fig. 9)*

In this experiment, we vary the maximum overlap factor (Ω_{max}) for different input rate values. Specifically, we set the input rate to 100, 1K, 10K, 100K and 1M tuples/sec. For all cases, the slide skewness was 0.6, and the number of ACQs was 2000. Recall that the overlap factor is the ratio between an ACQ's range and its slide. Hence, increasing the Ω_{max} increases the number of final-aggregations, but has no effect on partial-aggregation.

In Figure 9 we plot the normalized cost of *Weave Share* to *Shared*. As expected, as Ω_{max} increases from 50 to 2000, the gain provided by *Weave Share* increases. For example, consider the case of 100K tuples/sec. For small values of Ω_{max} (less than 200), *Weave Share* generates a single shared tree. For higher values of Ω_{max} (200 to 2000), *Weave Share* outperforms *Shared* by 19% to 74%. Recall that the overlap factor is multiplied by the edge rate in Eq.1, which is the cost of final-aggregation. Since *Weave Share* generates plans that consist of more than one tree, it keeps the maximum value of $E_i\Omega_i$ as small as possible. This is what enables *Weave Share* to outperform *Shared* for higher values of Ω_{max}.

7.1.4 *Slide Skewness (Fig. 10)*

In this experiment, we examine the slide distribution skewness parameter. By increasing the skewness, the query workload will contain more large-slide queries as generated by the Zipf distribution. Figure 10 shows the normalized cost of *Weave Share* to *Shared* for different number of ACQs, at input arrival rate of 100 tuple/second and maximum overlap factor of 10.

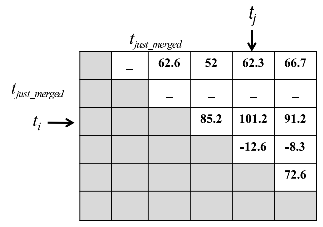
For all number of ACQs, we see that as the skewness increases, the relative gain provided by *Weave Share* increases. This continues until a global maximum gain is reached, where it starts to diminish until *Weave Share* performs similar to *Shared*. The reason is that initially, as the skewness increases the more large-slide ACQs we have, and hence the higher the *penalty* of sharing them with small-slide ACQs. *Weave Share* avoids this by selectively sharing ACQs that weave well together.

As the distribution becomes very skewed, most of the ACQs are large-slide ones, whereas small-slide ACQs gradually disappear. This means that grouping all in a single tree is the right choice. In this case, *Weave Share* captures this phenomenon and does share all ACQs. Figure 10 also shows that the more ACQs are in the system, the larger the maximum gain of *Weave Share* is.

7.2 Incremental performance

In this section, we study the performance of the *Incremental Weave Share* algorithm. For *Incremental Weave Share*, the tolerance factor is used to determine when to issue a reconstruction phase. A reconstruction phase is issued if the ratio of the current execution plan cost to the extrapolated cost using performance slope (given the learned plan-cost curve) exceeds the tolerance factor.

Figure 11 shows the overhead as number of comparisons versus the average relative error between the plan generated by *Incre-*



		t_{just_merged}			t_j	
	▨	_	62.6	52	62.3	66.7
t_{just_merged}	▨	▨	_	_	_	_
t_i	▨	▨	▨	85.2	101.2	91.2
	▨	▨	▨	▨	-12.6	-8.3
	▨	▨	▨	▨	▨	72.6
	▨	▨	▨	▨	▨	▨

Figure 12: Cost Lookup Table

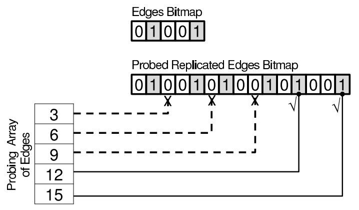


Figure 13: Edges Bitmap and Probing Process

mental Weave Share and the plan generated by the offline *Weave Share*, for different tolerance factor values (*the points' labels*). For instance, the point labeled as `Infinity` shows the incremental performance when no reconstruction is issued at all (tolerance = ∞). As expected, the figure shows that as the tolerance factor increases, the relative error increases while the overhead decreases. It also shows that the relative error is always less than or equal to the tolerance factor. From the above results, we conclude that a tolerance factor of 20% or 30% achieves a good balance between performance and overhead.

8. OPTIMIZING THE OPTIMIZER

Given a set of n ACQs, the time complexity of *Weave Share* algorithm is asymptotically $O(n^3)$. The algorithm starts with n trees and in each iteration it reduces the number of trees by one. Thus, in worst case, the algorithm needs n iterations. In each iteration i, $(n-i)^2$ comparisons are needed to find the pair of trees that yield the maximum benefit. Thus, the total time complexity is $O(n^3)$.

Computing the benefit of merging two trees (say t_x and t_y), requires calculating the new edge rate $E_{x,y}$ (Eq. 5). Given that there is no closed-form formula that determines the common edges as as discussed in Section 4.4, this is clearly an expensive operation which requires counting the set of common edges between the two trees, t_x and t_y.

Conceptually, to calculate the new edge rate resulting from merging t_x and t_y into tree $t_{x,y}$, we need to extend the steps needed for merging two ACQs (described in Section 4.1) as follows:

1. Set the composite slide $CS_{x,y}$ to be the least common multiple of the individual slides of all ACQs in t_x and t_y.
2. The edge count M'_x of the ACQs in t_x within the new composite slide $CS_{x,y}$ is computed as: $M'_x = M_x \frac{CS_{x,y}}{CS_x}$, where the last term is the number of times CS_x has been replicated. Similarly, the edge count M'_y of the ACQs in t_y is computed.
3. The composite edge count $M_{x,y}$ is computed as: $M_{x,y} = M'_x + M'_y - M_c$.

In order to compute the last step, we need to know the number of common edges in the composite slide (M_c) between t_x and t_y. This could be done by checking each edge in each ACQ in t_y to see if it is the same to any edge of any ACQ in t_x. Each one of those checks requires two comparisons. Specifically, to check if edge e of some ACQ in t_y is the same to some edge of ACQ q_i in t_x, we check if e is divisible by s_i, or if $e - g_{i,1}$ is divisible by s_i, as illustrated in the following example.

EXAMPLE 4. *Consider a tree with one query q_x that has slide $s_x = 5$ and fragments $g_{x,1} = 2$ and $g_{x,2} = 3$. Further consider a query q_y which has slide $s_y = 3$ and fragments $g_{y,1} = 0$ and $g_{y,2} = 3$. If q_x and q_y are to be merged, the common slide length is $CS_{x,y} = 15$, the edge counts of stretched q_x and q_y are $M'_x = 6$ and $M'_y = 5$, respectively. Hence, $M_{x,y}$ is 5 plus 6 minus the number of common edges (M_c), which is computed by checking each and every edge of q_y against those of q_x.*

The first edge in q_y is $e = 3$, which is not divisible by the slide of $s_x = 5$ nor is $e - g_{x,1} = 3 - 2 = 1$ divisible by $s_x = 5$. Hence, it is not a common edge and $M_{x,y}$ is kept at 11 edges. The current edge e is then advanced to next edge $e = 6$, and the two comparisons are performed and so on until $e = 12$, where $e - g_{x,1} = 12 - 2 = 10$ is divisible by $s_x = 5$, i.e., it is a common edge and the count is decremented. Similarly, at $e = C_{x,y} = 15$, e is divisible by s_x and the count is decremented once again.

This naive approach encounters a high overhead given that counting the edges process is repeated many times in the main loop of the algorithm, where, in each iteration, an edge count is needed for each pair of trees. We propose three optimizations that can dramatically minimize this overhead as discussed next.

8.1 Optimization I: Cost Lookup

The first optimization is to memoize the benefit (i.e., cost reduction) gained by merging two trees in a two dimensional array called *Cost Lookup* table. Thus, in the main loop of the algorithm, only the first iteration will compute the cost saving for each pair of trees. Next iterations will use the lookup table for all pairs, except those that involve the new merged tree from the previous iteration. Thus, the number of computations in each iteration i is reduced from $(n-i)^2$ to $(n-i)$ computations. This minimizes the number of pairs for which an edge count needs to be performed.

Figure 12 shows a possible instance of the Cost Lookup table. To check if merging two trees t_i and t_j is beneficial or not, we lookup the entry $Cost\_Lookup[i][j]$, which is 101.2 in this instance. This means that merging t_i with t_j would reduce the cost by 101.2 operations per second. Negative values mean that the merge would actually increase the cost. $t_{justmerged}$ is the merged tree in a previous iteration and that is why all its entries are nullified in order to be recomputed.

8.2 Optimization II: Edges Bitmap

The second optimization is to use a bitmap vector that acts as a hash table to represent the edges. The top part of Figure 13 shows the bitmap vector for an ACQ q_x with $s_x = 5$ and edges at locations 2 and 5 (i.e., fragments $g_{x,2} = 2$ and $g_{x,1} = 3$). Given the *Edges Bitmap* structure, finding the common edges between two trees requires to simply traverse the edges of one of the Edges Bitmap to probe the other, i.e., check if they exist in the other bitmap. This requires a number of probes equal to the number of edges in one of the trees, regardless of the number of ACQs in the other tree. Effectively, this optimization pre-computes and materializes the results of finding the common edges described in Example 4.

The Edges Bitmap is maintained as follows. When the tree has one query at most two edges are hashed into the bitmap. When adding a query to a tree, 1) new bitmap is created with length equal

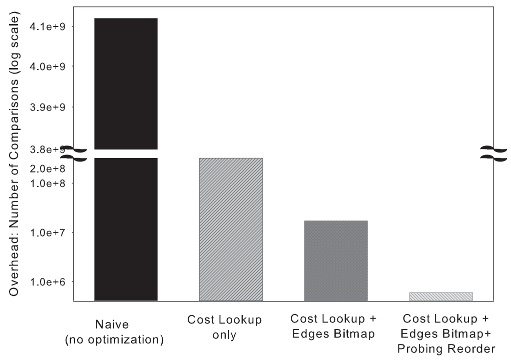


Figure 14: Optimizations' Benefits

to the new composite slide, 2) the old bitmap is replicated in this new bitmap and the previous count of edges is updated accordingly, and 3) the edges of the new query are hashed into the new bitmap, incrementing the edge counter only if no collision occurs.

8.3 Optimization III: Probing Reorder

Clearly, given the Edges Bitmap structure, the overall complexity of the algorithm will be affected by the choice of which bitmap to probe when counting common edges. Similar to join optimization, which uses the relation with fewer blocks to probe the other, we propose to use the tree with fewer edges (i.e., smaller edge rate) to probe the other. (this is illustrated in the lower part of Figure 13, where we used q_y which has 5 edges to probe q_x which has 6 edges). Specifically, the bitmap of the probed tree is replicated to the new composite slide, while the bitmap of the probing tree is used to generate an array of edges in the new composite slide. Edges in the array are then hashed into the bitmap of the probed tree, and if collision occurs, then the checked edge is common.

8.4 Impact of Optimizations

The impact of the above optimizations can be seen in Figure 14, where we report our run of *Weave Share* and its optimized versions for an experiment with 250 ACQs, input rate of 100 tuple/second, a slide skewness of 0.7 and a maximum overlap factor 10; we obtained similar results for different workload settings. The figure shows the overhead of the naive *Weave Share*, where no optimization is used, compared to the three optimization variants. In the first variant, only cost lookup is used. In the second variant, both cost lookup and edge bitmap are used and finally, in the third variant all three optimizations are used. Figure 14 (notice the log scale for the Y-axis) shows that each proposed optimization technique roughly adds one order of magnitude improvement over the naive implementation. Overall, with all three optimizations, *Weave Share* executes three orders of magnitude more efficiently.

9. CONCLUSIONS

In this paper, we introduced the concept of *Weaveability* of aggregate continuous queries (ACQs) that captures their potential for shared processing. We proposed *Weave Share*, a new cost-based multiple ACQs optimizer that selectively groups ACQs into multiple execution trees considering all cost factors, to minimize total cost. We also proposed *Incremental Weave Share* that supports dynamic query optimization when ACQs are added or deleted on demand. We experimentally evaluated and performed sensitivity analysis on the quality of the query plans generated by the *Weave Share*. The results show that *Weave Share* outperforms all alternative schemes. Finally, we developed practical implementation of *Weave Share* and experimentally illustrated their gains.

Acknowledgments This work was supported in part by NSF awards IIS-0534531 and IIS-0746696, an Andrew Mellon Predoctoral Fellowship, and Australian Research Council award DP110102777.

10. REFERENCES

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.
[2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
[3] A. Arasu et al. STREAM: The stanford stream data manager. In *SIGMOD*, 2003.
[4] E. Bach, K. Pruhs. Personal communications, June 2010.
[5] M. R. Garey, D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. WH.Freeman & Co., 1990.
[6] T. M. Ghanem et al. Incremental evaluation of sliding-window queries over data streams. *TKDE*, 19(1):57–72, 2007.
[7] L. Golab et al. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, 2006.
[8] G. Graefe, W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
[9] M. A. Hammad et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.
[10] R. Huebsch et al. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.
[11] R. Johnson et al. To share or not to share? In *VLDB*, 2007.
[12] S. Krishnamurthy et al. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
[13] J. Li et al. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 2005.
[14] J. Li et al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.
[15] S. Madden et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
[16] H. Mistry et al. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, 2001.
[17] K. Naidu et al. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.
[18] P. Roy et al. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
[19] W. Scheufele, G. Moerkotte. On the complexity of generating optimal plans with cross products. In *PODS*, 1997.
[20] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.
[21] Streambase, http://www.streambase.com, 2006.
[22] System S, http://domino.research.ibm.com/, 2008.
[23] S. D. Viglas, J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.
[24] S. Wang et al. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *VLDB*, 2006.
[25] R. Zhang et al. Multiple aggregations over data streams. In *SIGMOD*, 2005.
[26] R. Zhang et al. Streaming multiple aggregations using phantoms. *VLDBJ*, 19(4):557–583, 2010.
[27] J. Zhou et al. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.