2012 IEEE 28th International Conference on Data Engineering

Three-level 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
{*shenoda, panos, labrinid*}*@cs.pitt.edu*

[2]*School of Information Technology and Electrical Engineering*
The University of Queensland
m.sharaf@uq.edu.au

***Abstract*—Aggregate Continuous Queries (ACQs) are among the most common Continuous Queries across all classes of monitoring applications and typically have a high execution cost. As such, optimizing the processing of ACQs is imperative for Data Stream Management Systems to reach their full potential. Existing multiple ACQs optimization schemes focus on ACQs with varying window specifications and pre-aggregation filters and assume a processing model where each ACQ is computed as a final-aggregation of a sub-aggregation. In this paper, we propose a novel processing model for ACQs, called *TriOps*, that minimizes the repetition of operations at the sub-aggregation level, and a new multiple ACQs optimizer, called *TriWeave*, that is *TriOps*-aware. We analytically and experimentally demonstrate the performance gains of our proposed schemes, showing their superiority over alternative schemes. Finally, we generalize *TriWeave* to incorporate the classical subsumption-based multi-query optimization techniques for handling overlapping group-by attributes.**

I. Introduction

Streams Aggregation. Aggregate Continuous Queries (ACQs) are among the most common Continuous Queries across all classes of monitoring applications (e.g., [16], [15], [21]). Typically, many ACQs monitor the same data input stream. In fact, more than often, these ACQs are also computing the exact same aggregate function, but may have slightly different specifications, such as the window specifications, pre-aggregation filters (i.e., predicates), and group-by attributes.

For example, a network monitoring application could employ three ACQs to monitor the IP traffic, all of which could compute the *COUNT* of incoming packets. While the first ACQ could report the count in the last minute, updated every five seconds, the second and third ACQs could report the count in the last minute, to be updated every half minute. Further, the first ACQ might be interested in the count of IP traffic originating from a specific source, i.e., have a predicate that the source IP has a certain value. The second and third ACQs, on the other hand, might be counting all received packets, but have them grouped by source IP and destination IP, respectively.

While many ACQs, like the three ACQs in our example above, compute the same aggregate function over the same input data steam, they have different specifications, depending on the user and the purpose of the ACQ.

Motivation. Given the cost and commonality of ACQs, optimizing their processing is crucial in order for Data Stream Management Systems (DSMSs) (e.g., [2], [4], [5], [13], [6], [7], [27], [25], [26]) to achieve the scalability needed to handle the typical large volumes of data and large numbers of ACQs.

This need has motivated the development of several techniques for the efficient processing of ACQs, which could be broadly classified into techniques for: 1) the *implementation* of the continuous aggregation operator, and 2) the *multi-query optimization* of multiple ACQs.

Under the first set of techniques (i.e., operator implementation), *partial aggregation* has been proposed to minimize the repeated processing of overlapping data windows within a single aggregate (e.g., [17], [18], [16], [9]). In particular, partial aggregation aims at processing each input tuple only once and assembling the final aggregate value from a set of partial aggregate values. Specifically, ACQ processing is modeled as a two-level (i.e., two-operator) query execution plan: in the first level a *sub-aggregate* function is computed over the data stream generating a stream of partial aggregates, whereas in the second level a *final-aggregate* function is computed over those partial aggregates. We refer to this two-level aggregation processing model as *TwoOps* hereafter.

Under the second set of techniques (i.e., multi-query optimization), the general principle is to minimize (or eliminate) the repeated processing of overlapping operations across multiple ACQs. This repetition occurs as a result of processing the same data by different queries, which exhibit an overlap in at least one of the following specifications: 1) predicate conditions, 2) group-by attributes, or 3) window settings.

On one hand, leveraging overlaps in predicate conditions and group-by attributes across different queries has been the focus of intensive research on traditional multi-query optimization, which typically relies on the detection of common subexpressions. On the other hand, the introduction of window-based continuous queries for the processing of statefull operators (i.e., joins and aggregates) over unbounded data streams has motivated recent research on the shared processing of queries with overlapping windows.

For instance, the *Shared Time Slices* technique [16] has been proposed to share the processing of multiple ACQs with varying windows. It has also been extended into *Shared*

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.112

Data Shards in order to share the processing of varying predicates, in addition to varying windows. Orthogonally, the *Intermediate-Aggregates* optimizer [21] extends classical subsumption-based multi-query optimization techniques towards sharing the processing of multiple ACQs with varying group-by attributes and similar windows. Regardless of the differences between the above multi-query optimization techniques, they all rely on the same underlying *TwoOps* implementation of the aggregate operator.

Similarly, our cost-based *Weave Share* multiple ACQs optimizer [12], realized in the AQSIOS prototype [6], [3], adopts *TwoOps*. Like *Shared Time Slices*, *Weave Share* addresses the problem of shared processing of ACQs with varying windows. *Weave Share*, however, employs a novel metric based on the concept of *Weaveability* that allows the optimizer to selectively partition the ACQs workload into multiple disjoint execution trees resulting in a dramatic reduction in the total cost of processing the final-aggregation operators in those trees.

Under *TwoOps*, however, partitioning of ACQs requires duplicating the sub-aggregate level operation across the different disjoint trees (i.e., one for each tree). Naturally, *Weave Share* considers that duplicated cost in its optimization objective and tries to minimize the number of generated trees to minimize the overall cost. Nevertheless, the advantage of using *Weave Share* could be significantly restricted by this trade-off between the reduction in final-aggregation with the increase in sub-aggregation due to the use of multiple sub-aggregate operators. This suggests the need for a new underlying aggregate operator implementation in order to fully reap the benefits of *Weave Share*.

Contributions. In this paper, we effectively address this need by proposing to replace the *TwoOps* model with a three-level one that provides the advantage of sharing the sub-aggregation operator across all trees, while at the same time allowing the utilization of *Weaveability* at the final-aggregation operators as employed by *Weave Share*. In particular, the contributions of this paper are as follows:

1) *TriOps*, a novel three-level aggregation processing model, that uses an intermediate operator between sub- and final-aggregation to minimize the repetition of operations at the sub-aggregate level.
2) *TriWeave*, a new *Weave Share* optimizer, that works in synergy with *TriOps* to minimize the total cost of processing multiple ACQs.
3) An experimental evaluation study that demonstrates the performance gains provided by *TriWeave* and shows that *TriWeave* is superior to other alternatives.

In addition to performance gains, *TriOps* still maintains the attractive features of the *TwoOps* model, which allow it to incorporate classical multi-query optimization techniques for exploiting overlapping predicates and group-by attributes. As such, the fourth contribution of this paper is:

4) *GTWeave*, a generalization of *TriWeave* that integrate the classical subsumption-based multi-query optimization techniques (i.e., overlapping predicates and group-by attributes) with the new weaveability-based multi-query optimization (i.e., *Weave Share*).

Outline: In Section II we provide the necessary background. We then present *TriOps* in three stages. First, we consider the case of varying windows in Section III; then the case when predicates are also different in Section IV; and finally, the general case when all specifications are different in Section V. We present the experimental evaluation of our proposed schemes in Section VII. We discuss the related work in Section VIII and conclude the paper in Section IX.

II. Background

In this Section, we first provide the necessary background on data streams aggregation in Section II-A. We then describe the *Paired Window* technique in Section II-B and the shared processing schemes that utilize it in Section II-C.

A. Streams Aggregation

An ACQ is defined over a window 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., $r = 1$ hour) and update it (i.e., compute a new average) every 30 minutes (i.e., $s = 30$ min). The range and slide intervals could be defined either based on the number of tuples or based on time. We consider the more general time-based definition for both the range and slide; however, our contributions are applicable to the tuple-based definition as well.

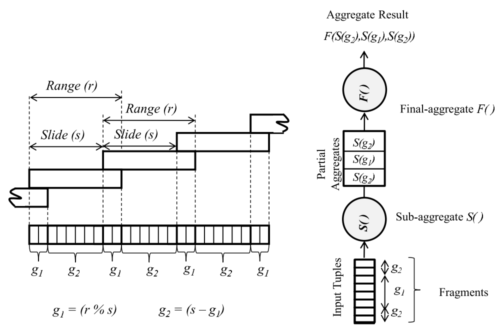
Producing a new aggregate result per window requires processing each tuple within the window range. The slide, on the other hand, defines how the window boundaries move over the input stream. For instance, when the slide is less than the range (*sliding window*), different consecutive windows overlap 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 it to be updated every ten minutes. The user registers an ACQ with $r = 1$ hour 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 (from that time-point). Hence, each input tuple is participating in the aggregate computation of six consecutive windows (1 hour / 10 min = 6).

In a straightforward implementation of ACQs, input tuples are buffered and once a boundary line 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.

B. Paired Window Technique

In order to implement aggregate continuous operators efficiently, current techniques exploit *partial aggregation* to minimize the repeated processing of overlapping windows (e.g., [17], [16]). In particular, each input tuple is processed only once; the final aggregate value is assembled from a set of partial aggregate values. Specifically, an ACQ processing is modeled as a *TwoOps* (i.e., two operators) query execution plan: in the first level a sub-aggregate function is computed



(a) Fragments (b) Plan

Fig. 1. Paired Window Technique

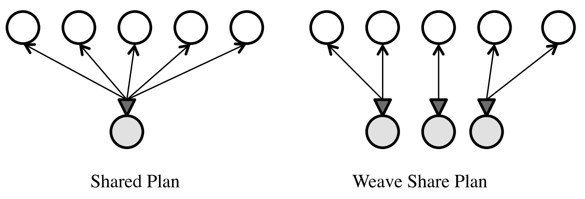


Fig. 2. Shared Plan vs. Weave Shared Plan

over the data stream, which generates a stream of partial aggregates, whereas in the second level a final-aggregation operator function is computed over those partial aggregates.

For example, under the partial aggregation scheme, an ACQ `COUNT(*)` query is computed using (1) a `COUNT(*)` on each sub-window 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 P over each dataset g_i and a *final-aggregate* function F over the partial aggregates. Formally,

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

Partial aggregation reduces the processing cost by processing each input tuple only once by the sub-aggregate operator. 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. This observation has been utilized in the *Paired Window* technique [16] (Figure 1), which 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 $\frac{2r}{s}$ 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*, which is a timestamp where the tuples in g_i are assembled into a partial aggregate.

C. Shared Processing of ACQs

Based on the *Paired Window* technique, several query processing models and optimizers have been proposed for the shared processing of multiple ACQs. Specifically, the *Shared Time Slices* model (or for brevity *Shared*) has been proposed to share the processing of multiple ACQs with varying windows [16]. The main idea underlying *Shared* is to share the sub-aggregation process and generate fine grained fragments that satisfy all the different windows specified in the shared ACQs. In particular, under *Shared*, all ACQs are simply merged into one execution tree and share the same sub-aggregation operator (as shown in Figure 2). The shared sub-aggregation operator is thus responsible for generating fine grained fragments (i.e., partial results) that allow each of the shared ACQs to reassemble its final aggregation results according to its own range and slide parameters.

The sharing of the fragments among a set of n ACQs $Q = q_1, q_2, ..., q_n$ is achieved by defining a *composite slide* to be the least common multiple of the individual slides of all ACQs, i.e., $CS = lcm(s_1, ..., s_n)$. 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 copied and repeated to the length of s'_i (=repeated $\frac{s'_i}{s_i}$ times). 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).

Orthogonally, the *Weave Share* [12] optimizer has been proposed to utilize the *Paired Window* technique in order to minimize the cost of the final aggregation. *Weave Share* selectively groups the ACQs into multiple execution trees. Under *Weave Share*, each tree employs its own sub-aggregation operator, which is shared between the ACQs in that tree (as shown in Figure 2). In order to decide the best grouping of ACQs, *Weave Share* introduces the concept of *weaveability* [12], which measures the potential gains from the shared processing of ACQs and *weaves*, i.e., groups together, the ACQs based on their *weaveability* measures.

In contrast to *Shared*, under *Weave Share* the sub-aggregation operator in each tree typically produces coarser grain fragments, which are closer in length to the ones to be produced if each ACQ were to be processed alone without sharing. This has the advantage of reducing the amount of final-aggregate operations compared to *Shared*. However, one challenge in the design of *Weave Share* is to balance this trade-off between the reduction in final-aggregation with the increase in sub-aggregation due to employing multiple sub-aggregate operators (one for each tree). In this paper, we effectively address that challenge by proposing *TriOps* to replace *TwoOps* with a three-level model that allows the sharing of the sub-aggregation operator across all trees (as in *Shared*), while at the same time maximizing the granularity of the produced fragments that are processed by the final-aggregation operators (as in *Weave Share*).

III. *TriOps* AND *TriWeave*

In this section, we first present our running example in Section III-A and then introduce *TriOps*, a new three-level processing model for multiple ACQs in Section III-B. Then we analyze its performance in Section III-C. Finally, we propose a new weaveability-based multi-query optimizer, called *TriWeave*, which assumes the *TriOps* processing model, in Section III-D.

A. Running Example

For the remainder of this paper, we will use four ACQs to illustrate our proposed schemes. Similarly to all previous works in optimizing the execution of ACQs [16], [31], [21], [12], we assume that each of these ACQs is not part of a larger complex CQ and is only characterized by pre-aggregation predicates, group-by attributes and window settings. The predicates can be arbitrary complex.

The specifications for the four ACQs q_a, q_b, q_c and q_d are summarized in Table I. As we incrementally build our proposed *TriOps* model and our *TriOps*-aware optimizers, we will use subsets of these ACQs and specifications accordingly.

TABLE I
RUNNING EXAMPLE QUERIES

ACQ	Window (r, s)	predicate	group-by attribute
q_a	(8, 5)	c_a	A
q_b	(5, 4)	c_b	BC
q_c	(10, 1)	c_c	AC
q_d	(5, 4)	c_d	CD

B. TriOps *Processing Model*

TriOps is a new aggregate operator implementation that works in synergy with the new *Weave Share* optimizer [12] to minimize the total cost of processing multiple ACQs. *TriOps* employs a three-level data processing model that minimizes the repetition of operations at the sub-aggregation level.

Let us first consider the case when different ACQs have varying windows specifications, but the same predicates and same group-by attributes (the cases with different predicate and group-by attributes are discussed next in Sections IV and V, respectively). As with all partial-aggregation based processing models, *TriOps* uses a sub-aggregation operator to aggregate input tuples once, generating a stream of fragments. In *TriOps*, a single sub-aggregation operator is shared among all ACQs. However, instead of directly rolling up the generated fragments into the final-aggregation operators, *TriOps* introduces a new intermediate level of aggregation.

In particular, the *intercede-aggregation* operator is introduced to the query plan between sub- and final-aggregation levels (as shown in Figure 3). This new level of aggregation is made aware of the weaved plan and its weaved groups (or equivalently trees). In particular, it behaves for each weaved group of ACQs as its unshared sub-aggregation operator in the case of *TwoOps*. In this way, *TriOps* avoids the disadvantages of replicating the sub-aggregation operator for each weaved group. Meanwhile, it also avoids the disadvantages of using a single sub-aggregation that is shared by all ACQs. That is, by utilizing a single sub-aggregation, *TriOps* avoids processing input tuples multiple times, and by making the *intercede-aggregation* operator aware of the weaved groups, it avoids the increase in the processing overhead, i.e., the number of aggregate operations needed at the final-aggregation level.

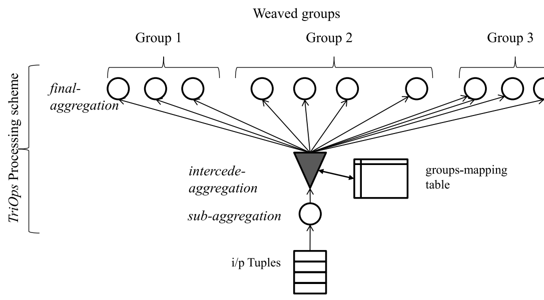


Fig. 3. *TriOps* Shared Processing Scheme

Specifically, the *intercede-aggregation* operator performs the following tasks:

1) It buffers all the fine-grain fragments generated by the sub-aggregation operator until they are rolled up into all their respective weaved groups.
2) It assembles the relevant fine-grain fragments into the ones expected by each weaved group and passes them to the group's final-aggregation operators, whenever a relevant edge of a group is reached.

Being aware of the weaved groups, the *intercede-aggregation* operator achieves the last step by coalescing, for each group, the smaller fragments generated by the single shared sub-aggregation operator into the stream of fragments that this group would have seen if it had its own sub-aggregation operator. This is done only once for each group, when an edge is due for one of the ACQs in that group. Thus, each fragment is aggregated once per group, instead of once per window instance as in the case with *TwoOps*. The following example illustrates the idea of *intercede-aggregation*.

Example 2: Consider the first two ACQs of our running example, namely, $q_a(8, 5)$ and $q_b(5, 4)$. For simplicity, let us assume that the weaved plan decides not to share the execution of those two ACQs. Thus, q_a has the following sequence of edges at timestamps: $3, 5, 8, 10, 13, ...$, whereas q_b has edges at timestamps $1, 4, 5, 8, 9, 12, ...$ Under *TriOps*, the shared sub-aggregation operator would produce fragments with the timestamps sequence of $1, 3, 4, 5, 8, 9, 10, 12, 13, ..,$ that is the union of the two sequences of edges. When the edge at timestamp 3 (of q_a) is reached, for instance, the *intercede-aggregation* operator will aggregate the fragments with timestamps 1 and 3 to produce the fragment that q_a is expecting and route this fragment to the input buffer of q_a. Similarly, when edge 4 (of q_b) is reached, the *intercede-aggregation* operator will aggregate the fragments generated at timestamps 3 and 4 to generate the fragment that q_b is expecting. This can be easily generalized to groups of ACQs where every edge belongs to a certain group, instead of a single ACQ, and the *intercede-aggregation* operator computes the fragment that this group expects to see.

A weaved plan using *TriOps* is illustrated in Figure 3. As shown in the figure, *intercede-aggregation* operator uses a *group-mapping* lookup table to generate the proper fragments for each group. This table is generated and maintained by the multi-query optimizer as will be explained in Section III-D.

C. TriOps *Cost and Advantages*

In this section, we analyze the cost function of a weaved plan using *TriOps* and discuss its advantages compared to *TwoOps*. Consider *TwoOps* where each weaved group of ACQs forms a tree rooted at a shared sub-aggregation operator. 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 trees $t_1, t_2, ..., t_m$ where all ACQs of a tree t_i are shared, then the cost, in terms of total number of aggregate operations per second, of each tree t_i is computed by:

$$C_{t_i} = \lambda + E_i \Omega_i \tag{1}$$

where λ is the data input rate and for a tree t_i, E_i is the edge rate (i.e., number of fragments the sub-aggregation operator of this group generates per second) and Ω_i denotes the total number of final-aggregation operations performed on each fragment. We refer to Ω_i as the *tree overlap factor* or *overlap factor* for short. For a set of shared ACQs $SQ = q_1, q_2, ..., q_k$ in a tree t_i, Ω_i is computed as:

$$\Omega_i = \sum_{j=1}^{k} \frac{r_j}{s_j} \tag{2}$$

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

$$C_{\textit{weaved plan, 2-operator}} = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{3}$$

Note that the first term of Equation 3 is the cost at the sub-aggregation level, whereas the second term is the cost at the final-aggregation level.

Equation 3 shows that the cost of grouping ACQs depends partially on the edge rate of each weaved group (E_i) which depends on the weaveability of the grouped ACQs. The *Weave Share* optimizer [12] utilizes *weaveability* to group the ACQs into a set of trees in a way that minimizes the total cost of the query plan, i.e., Equation 3. That is, to strike a balance between the two components of the cost function. In particular, *Weave Share*'s 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.

Given the *TriOps* model, however, the total cost of a weaved plan in Equation 3 becomes:

$$C_{\textit{weaved plan, TriOps}} = \lambda + m.E + \sum_{i=1}^{m} E_i \Omega_i \tag{4}$$

where E represents the edge rate of the shared sub-aggregation, and E_i is the edge rate that each weaved group receives from the *intercede-aggregation* operator. The term $m.E$ represents the cost of the *intercede-aggregation*, where each fragment is aggregated once for each group.

Comparing the cost function of *TriOps* (Equation 4) to that of *TwoOps* (Equation 3), our new processing scheme reduces the sub-aggregation cost from $m\lambda$ to $\lambda + m.E$, which is the cost of the sub-aggregation plus that of the *intercede-aggregation* operator. Since the edge rate E is typically much smaller than λ, *TriOps* typically reduces the cost by a factor proportional to $\frac{E}{\lambda}$. The only exception is the extreme case when the input rate is one tuple per time unit and the sub-aggregation is generating one fragment per time unit (i.e., $E = \lambda = 1$).

In addition to reducing the cost of the weaved plan, *TriOps* offers several other performance advantages, namely, efficient adaptivity, smaller operator invocation overhead, and less memory overhead.

Adaptivity becomes easily achievable because *TriOps* effectively defines a canonical “template” for a multiple ACQ plan. In particular, for the same set of ACQs, any weaved plan will share that same template and the only difference between those plans would be the weave grouping of the operators at the final aggregation level. Hence, to switch from one plan to another, for instance, as a result of a change in input rate, it is enough to change the group-mapping table, used by the *intercede-aggregation* operator to correctly direct the partial-aggregation results, to reflect the new grouping of final-aggregation operators. Further, the addition and deletion of ACQs becomes as simple as adding or dropping a final-aggregation operator, and updating the group-mapping table.

In terms of operator invocation overhead, *TriOps* replaces m sub-aggregation operators of *TwoOps* by exactly two operators: one shared sub-aggregation and one *intercede-aggregation*.

Finally, in terms of memory efficiency, given that *TriOps* uses a single sub-aggregation operator, input tuples are buffered until they are consumed only once, as opposed to being buffered until they are consumed m times, once per group, as in *TwoOps*. While the *intercede-aggregation* requires extra buffering of the fragments, the savings from shorter buffering of the input tuples surpasses this overhead. Specifically, instead of buffering λ tuples/second until they are consumed by all m sub-aggregation operators, the λ tuples/second are buffered until they are consumed once, and E fragments/second are buffered until they are consumed by the m groups.

D. TriWeave *Optimizer*

The fact that our new processing model reduces the cost of partial-aggregation suggests that a selective grouping of ACQs based on *TriOps* would result into more weaved groups and lead to better performance. This prompted us to develop *TriWeave*, which is a new *TriOps*-aware multiple ACQ optimizer.

The *TriWeave* optimizer works similarly to the *Weave Share* optimizer [12], trying to selectively group together the ACQs that weave well. That is, to group ACQs in a way that minimizes the total weaved plan cost as per Equation 4. The steps of *TriWeave* are shown in Algorithm 1 and can be

Algorithm 1 The *TriWeave* Algorithm

```
1: Input: A set of $n$ ACQs
2: Output: TriWeave query plan $P$
3: begin
4: $P \leftarrow$ Create a weaved group for each ACQ
5: $l \leftarrow n$
6: $(\textit{max-reduction}, t_1, t_2) \leftarrow (0, -, -)$ {group-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 $temp > \textit{max-reduction}$ 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 $\textit{max-reduction} > 0$ then
17:    merge($t_1$,$t_2$)
18:    $l \leftarrow l - 1$
19:  end if
20: until No merge is done
21: group-mapping $\leftarrow$ Generate-Mapping-Table($P$)
22: Return $P$
23: end
```

summarized as follows. First, the plan is initialized by creating a weaved group for each ACQ, i.e., no sharing at all. Then, as long as it is beneficial, i.e., reducing the total cost of the plan, the pair of groups that yields the maximum reduction in the plan cost when shared is merged and the plan is updated. When no such pair of groups is found, the group-mapping table is generated and the current plan is returned as the *TriWeave* plan.

Notice that based on Equation 4, the *TriWeave* optimizer relies on accurately calculating the E_is (i.e., the edge rate of each weaved group) in order to compute the cost reduction if two weaved groups are merged (Algorithm 1, line 10). Computing the $E_{x,y}$ of a weaved group resulting from the merging of two weaved groups t_x and t_y requires counting the number of common edges between the two weaved groups t_x and t_y to eliminate duplicate edges. Unfortunately, when merging two weaved groups and constructing their composite slide (see Section II-C), there is no closed-form formula that determines the common edges. However, the three optimization techniques to efficiently count the common edges proposed and evaluated in [12] for *Weave Share* are also applicable to *TriWeave*.

We experimentally demonstrate the performance gains of *TriWeave* in Section VII. The results confirm our hypothesis that *TriWeave* generates better quality weaved plans with more weaved groups compared to *Weave Share* and that these *TriWeave* multiple ACQs plans are superior to their alternatives.

IV. *TriOps*: Windows and Predicates

In this section, we study the case when ACQs have varying windows specifications as well as different predicates. We first briefly overview the *Data Shards* technique in Section IV-A, and in Section IV-B we provide the details on how *TriOps* efficiently adopts the *Data Shards* scheme to process ACQs with different predicates and varying windows.

A. Data Shards Technique

The *Data Shards* [16] technique was proposed to handle ACQs with arbitrary complex predicates as well as varying window specifications, assuming the *TwoOps* model. The main advantage of *Data Shards* is that it avoids any unnecessary repeated evaluation of predicates. In particular, under *Data Shards* each predicate is evaluated for each tuple exactly once in a pre-processing phase prior to the sub-aggregation level. As an outcome of this pre-processing phase, each tuple is augmented with a *predicates signature* which encodes the results of evaluating all the predicates for this tuple. This signature is basically a bitmap vector, where each bit represents a predicate and is set to one only if this predicate evaluates to true for this tuple. Thus, this signature identifies which set of ACQs this tuple belongs to.

Given the set of augmented tuples, the sub-aggregation operator then aggregates all tuples of identical signatures together, to produce a set of *fragment-signature* pairs. Once an edge is due, the signature of each fragment-signature pair is examined and the augmented fragment is forwarded onto the input buffer of the final-aggregation operator of every ACQ whose predicate is satisfied by that fragment.

The incorporation of the *Data Shards* technique in the *Weave Share* optimizer is straightforward. The tuple augmentation process, where each tuple is evaluated against all predicates and augmented with a predicates signature, is done as a pre-processing phase as in *Data Shards*. As part of the pre-processing phase each augmented tuple is pushed to all the sub-aggregation operators in the weaved plan. When a weaved plan is generated, each sub-aggregation operator is associated with a set of <predicates signature, ACQ> pairs that encodes the predicates of the ACQs in its weaved group. A sub-aggregation operator uses the predicates signature in its set of <predicates signature, ACQ> pairs to filter out tuples which do not satisfy the predicates of any ACQ in its weaved group and aggregate the rest into the fragment-signature pairs. It also uses the <predicates signature, ACQ> pairs to forward the augmented fragments to all the appropriate final-aggregation operators when they are due.

There are two drawbacks of the *Data Shards* scheme that the *TriOps* processing model addresses. The first drawback is the transient memory overhead involved in replicating the fine-grain fragments in the input buffer of the final-aggregation level. That is, given a set of l predicates, a signature of length l is augmented to each tuple, yielding 2^l different possible signatures. This means that each fragment is split into possibly 2^l fragment-signature pairs. Replicating these fragments in the input buffers of each and every ACQ linearly increases the memory overhead, whereas the memory overhead increases exponentially when adding new ACQs with new predicates. Directly related to this issue is the second drawback, which is the increase in the processing overhead. That is, the final aggregation operator of each ACQ needs to perform 2^l extra aggregations per fragment, for each window instance.

The two drawbacks mentioned above are further escalated when *Data Shards* is used in *Weave Share* due to the replication of sub-aggregation operators, which leads to further repli-

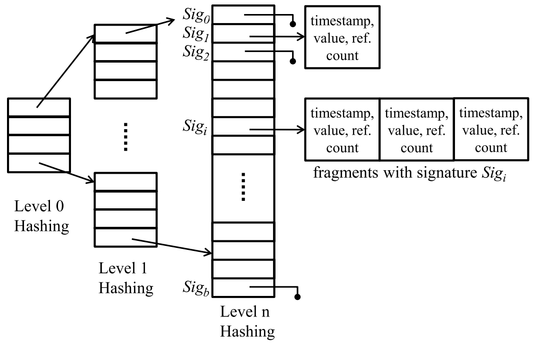


Fig. 4. Inverted Predicate Signatures Structure

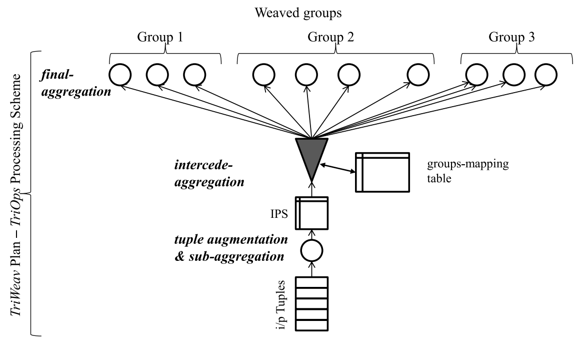


Fig. 5. *TriOps* - Windows and Predicates

cation of the augmented fragments as discussed above. *TriOps*, however, overcomes the drawbacks through the *intercede-aggregation* level and by fusing the tuple-augmentation with the sub-aggregation level as we discuss next.

B. TriOps*: Handling Different Predicates*

TriOps efficiently adopts the *Data Shards* scheme to process ACQs with different predicates as well as varying window specifications. To do so, *TriOps* first fuses the tuple-augmentation with the sub-aggregation phase. The goal of this merge of tasks is to eliminate the need to store the signature of each tuple, or fragment. Second, the *intercede-aggregation* operator, which already handles the routing of fragments, is made predicate-aware in order to properly coalesce and route the fragments to the final-aggregation level when they are due, eliminating the need to replicate the fragment-signature pairs in the input buffers of each query's final aggregation operator.

In particular, *TriOps* utilizes an inverted-predicate signatures (*IPS*) index to store and efficiently retrieve the fragments. *IPS* is essentially a multi-level hash-based structure shared between the sub-aggregation and the *intercede-aggregation* operators as shown in Figure 4.

Each signature entry Sig_i at level n of the *IPS* points to a linked list of nodes, where each node represents a fragment that satisfies signature Sig_i. Thus, signatures are no longer augmented to the fragments, but are instead embedded in the *IPS* structure. Further, to allow fusing the tuple-augmentation with the sub-aggregation phase, each fragment is represented by means of its *timestamp* and its aggregate *value*, which are two fields included in each node of the linked list.

Finally, to facilitate the routing of fragments to the final-aggregation level, a *reference count* field is added to each node, which is initially set to the number of weaved groups that are to read that fragment and once the reference count drops to zero, the fragment is discarded. Accordingly, each group in the group-mapping table is further augmented with a set of fixed pointers to entries in the *IPS* corresponding to the set of fragments that satisfy that group's predicates.

Figure 5 shows a *TriWeave* plan using *TriOps* for handling different predicates and windows. Given such plan, the execution proceeds as follows:

1) The sub-aggregation operator processes each input tuple and incrementally evaluates all the predicates (e.g., using predicate indexes and group filters [19]) on this tuple. The results of these predicate evaluations are used to locate the entry in the *IPS* to perform the aggregation in-place (i.e., updating the value field).
2) When an edge is due for a certain weaved group, the *intercede-aggregation* operator looks up the group-mapping table to directly collect the different fragments that belong to this group, aggregates them and produces the fragments of this group, discarding the expired fragments.
3) Finally, each final-aggregation operator aggregates the augmented-fragments that satisfy its predicate to generate the final results.

We illustrate the above steps with the following example.

Example 3: For simplicity, consider only the first two ACQs (i.e., q_a and q_b) of our running example, which have two different predicates: c_a and c_b, as shown in Table I. Further, assume for the purpose of this example that each ACQ forms its own weaved group. In this case, the signature has two bits, and there are three possible signature values: $01, 10$ and 11. Figure 6 shows a snapshot of the *IPS* for these two ACQs. Assuming that the most significant bit in the signatures represents predicate c_a, Figure 6 highlights the set of fragments that are to be aggregated when the edge with timestamp 3 of q_a is reached. Thus, the *intercede-aggregation* will aggregate these fragments and push them to the input buffer of the final-aggregation operator in q_a.

Figure 6 also shows several interesting possible cases. Specifically, in the "01" signature entry, the fragment contributing to the edge at timestamp 1 of q_b was already consumed and therefore deleted. Also, in the "10" signature entry, the fragment at timestamp 3 has its value field equals 0 because no tuples with this signature were inserted during the time interval covered by that fragment. Finally, in the "11" signature entry, Figure 6 shows that some tuples with this signature started to aggregate to form a new fragment with timestamp 5.

When adding new ACQs with new predicates, a new predicate is incorporated by introducing an extra bit as the most significant bit in the signature. Thus, all the previous signatures remain valid and the new signatures are hashed properly. Deletion of ACQs, however, is more involved. In particular, if the deleted ACQ results in deleting the predicate represented

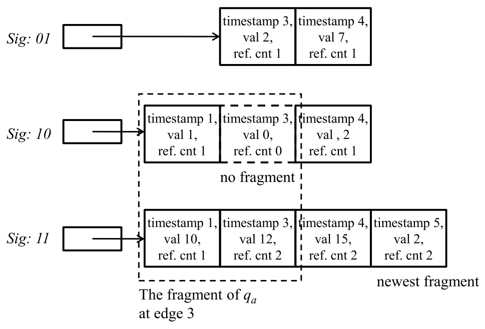

Sig: 01
timestamp 3, val 2, ref. cnt 1
timestamp 4, val 7, ref. cnt 1
Sig: 10
timestamp 1, val 1, ref. cnt 1
timestamp 3, val 0, ref. cnt 0
timestamp 4, val , 2 ref. cnt 1
no fragment
Sig: 11
timestamp 1, val 10, ref. cnt 1
timestamp 3, val 12, ref. cnt 2
timestamp 4, val 15, ref. cnt 2
timestamp 5, val 2, ref. cnt 2
The fragment of q_a at edge 3
newest fragment

Fig. 6. Shards that belong to the same fragment

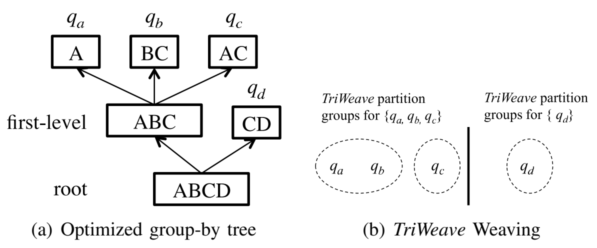

q_a
q_b
q_c
A
BC
AC
q_d
first-level
ABC
CD
root
ABCD
(a) Optimized group-by tree
TriWeave partition groups for $\{q_a, q_b, q_c\}$
TriWeave partition groups for $\{q_d\}$
q_a q_b
q_c
q_d
(b) *TriWeave* Weaving

Fig. 7. An Instance of Four ACQs

by the most significant bit, then the *IPS* table can be reduced to half. If that is not the case, then a new *IPS* with half the size is instantiated and the two *IPS*s work simultaneously until no entries exist in the old *IPS* at which point it is to be discarded and completely replaced by the new one.

V. *TriOps*: Windows and Group-by

In this section, we demonstrate how *TriOps* can efficiently optimize the processing of multiple ACQs with varying window specifications, and group-by attributes. We first briefly overview the *Intermediate-aggregates* scheme [21] in Section V-A and then discuss how *TriOps* utilizes that *Intermediate-aggregates* scheme to optimize ACQs which have the same predicate, but varying window specifications and different group-by attributes in Section V-B.

A. *Intermediate-aggregates*

In [30], [31], [21], the problem of optimizing multiple ACQs with different group-by attributes was addressed in the context of Gigascope's two layer architecture [8]. At the lower layer, which can be viewed as a sub-aggregation level, an aggregation is processed using a hash table consisting of a specified number of entries, each of which is a <group-attribute, value> pair. Since entries are fixed, multiple groups may hash to the same entry and a collision occurs when a new tuple r hashes to an entry b_k and r does not belong to the same group as the existing group in b_k. When a collision occurs, the current entry in b_k is evicted and a new group corresponding to r is created in b_k. The evicted entry is passed to the upper layer, that can be thought as a final aggregation layer, to be aggregated with the previously evicted entries of the same group when the aggregation result is due.

Given multiple ACQs with different group-by attributes, in order to minimize the cost of probing multiple hash tables for every new tuple as well as the eviction rate to the upper layer, in [30], [31] the idea of *Phantoms* was introduced. *Phantoms* are essentially a set of sub-aggregates, each of which shares the processing with a group of ACQs which can be derived from the *Phantom*. For instance, for a group of three ACQs defined on attributes A, B and C, the *Phantom* might maintain a hash table for ABC. When a new record arrives, instead of probing three hash tables A, B, and C, only the hash table ABC is probed and the hash tables A, B, and C are probed when an entry is evicted from ABC.

The *Intermediate-aggregates* [21] generalizes *Phantoms* in a hierarchy and the *Intermediate-aggregates* optimizer aims at generating a group-by tree (see Figure 7(a) of our running example) with minimal cost, subject to a memory constraint, where the cost is the cost of hashing. In the context of our paper, the hashing cost can be mapped to the number of aggregation operations. The group-by tree, similar to the query plan, determines the computation flow and sharing between the ACQs. Specifically, each node is labeled with the set of group-by attributes that will be used to perform the aggregation in the corresponding operator. Each internal node in the tree represents a partial aggregation operator that is shared between its subsequent nodes (in our example node ABC), whereas leaf nodes represent final aggregation operators (in our example A, BC, AC, and CD). Edges of the tree determine the flow of partial aggregates. The root of the group-by tree always represents the input stream. The first-level could have a single node, which means it is a single partial aggregation that is shared among all the ACQs, or multiple nodes, which means there is no single partial aggregation shared among all ACQs. Whenever the tree has a single node in the first level, we consider it to be the root and its child nodes to be the first-level nodes, as it is the case in Figure 7(a).

Finding the optimal group-by tree was shown in [21] to be an NP-hard problem. *Intermediate-aggregates*, therefore, aims at minimizing the cost using a greedy heuristic approach. Briefly, *Intermediate-aggregates* starts with a simple group-by tree that shares all ACQs using a single partial aggregation. That is, the initial group-by tree has a single root, and all first level nodes are leaf nodes. Then, in each iteration, the heuristic considers simple modifications of the tree, e.g., adding an internal node or splitting a non-leaf node, and chooses the modification that leads to maximum reduction in the total cost. Notice that this greedy policy is very similar to that of *TriWeave* and *Weave Share* optimizers.

B. *TriOps: Handling different Group-by Attributes*

In order to optimize the shared processing of multiple ACQs with varying window specifications and different group-by attributes, we utilize the *Intermediate-aggregates* optimizer in the following manner. First, we apply the *Intermediate-aggregates* optimizer as if all windows were identical to generate the group-by tree corresponding to that case. Given this optimized group-by tree, each first-level node (i.e., a node that is a child of the root) represents a set of ACQs that can

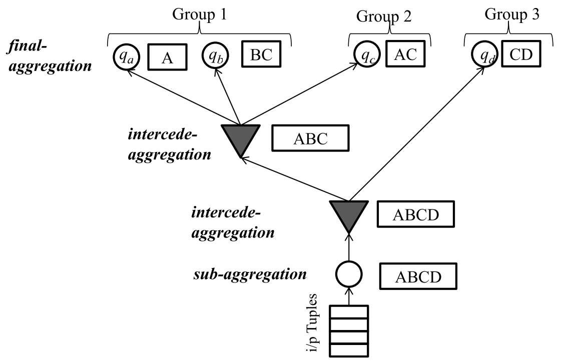


Fig. 8. Integrating *TriWeave* Plan with Intermediate-aggregates Tree

share their processing, given their different group by attributes, but assuming the same windows. We then apply *TriWeave* on each of these sets independently to further partition each set into weaved groups. Finally, we integrate the weaved groups with the group-by tree using *TriOps*. That last step is achieved by mapping each internal node in the group-by tree to an *intercede-aggregation* operator that is aware of the weaved groups, and can also perform a group-by aggregation using the set of attributes of that group-by tree node. To illustrate these steps, consider our running example (Figures 7 and 8).

Example 4: Assume we have four ACQs q_a, q_b, q_c and q_d with window specifications: $(8,5),(5,4),(10,1)$, and $(5,4)$, respectively. Assume also that the ACQs have group-by attributes: A, BC, AC, and CD, respectively. We first generate the group-by tree for the four ACQs, using the *Intermediate-aggregates* scheme, which is shown in Figure 7(a). The label of each node represents the set of group-by attributes used by this node. That is, each node represents an aggregation operator that performs a group-by aggregation using this set of attributes. The group-by tree in this case has one internal node labeled ABC. Thus, the set of leaf nodes (i.e., ACQs) of the sub-tree for which ABC is the root, represents a set of ACQs that share their processing given their different group-by attributes, but assuming they have the same window specifications. Specifically, q_a, q_b and q_c are shared together, while q_d is processed separately. Thus, we have two sets of ACQs, set $T_1 = q_a, q_b, q_c$ and set $T_2 = q_d$. We proceed by generating the weaved plan for each set, i.e., apply *TriWeave* on T_1 then on T_2. Figure 7(b) shows the output of this step which weaves T_1 into two groups, one that shares q_a and q_b, while the other has q_c by itself. The weaved plan of T_2 is trivial as it has one ACQ. The last step is to integrate the results of the first two steps together into a *TriOps* plan. Figure 8 shows the integrated *TriOps* plan. Simply, the root of group-by tree is mapped to the sub-aggregation operator, while each internal node is mapped into an *intercede-aggregation* operator.

The procedure described above follows a conservative approach towards sharing. Specifically, two ACQs are shared only if they are shared under both *TriWeave* and *Intermediate-aggregates*. For instance, while q_b and q_d have identical window specification, they do not belong to the same weaved group of the final *TriWeave* plan.

Algorithm 2 Generalized *TriWeave* Optimizer

```
1:  **Input: A set of** $n$ **ACQs**
2:  **Output: generalized *TriWeave* query plan** $P$
3:  **BEGIN**
4:  $T \leftarrow$ Generate the group-by optimzed tree
5:  $WP \leftarrow \emptyset$ {weaved plan}
6:  **for** For every node $t_i$ that is child of $root(T)$ **do**
7:      $C_i \leftarrow ACQs(t_i)$
8:      $P_i \leftarrow$ generate_TriWeave_plan($C_i$)
9:      $WP \leftarrow P_i \cup WP$
10: **end for**
11: $P \leftarrow$ integrate($T$, $WP$)
12: $P \leftarrow$ augment_IPS($P$)
13: Return $P$
14: **END**
```

Notice that the group-by tree might have multiple levels of nodes. For instance, in the above example q_a and q_c could have a common parent node labeled AC, which performs a group-by aggregation using the attributes AC and is not a first-level node. Mapping such an internal node AC to the *TriOps* plan depends on the outcome of applying *TriWeave* on the set of ACQs rooted at AC (i.e., q_a and q_c). Following the same conservative approach towards sharing, if the weaved plan shares q_a and q_c, then AC is mapped into another *intercede-aggregation* operator. Otherwise, it is just dropped from the integrated plan.

VI. *GTWeave*: Generalized *TriWeave* Optimizer

In this section we put it all together and present *GTWeave*, a generalized weaveability-based optimizer. *GTWeave* integrates the techniques we have proposed in the previous sections towards generating optimized plans for the efficient processing of multiple ACQs with different window specifications, different predicates and different group-by attributes.

Basically, *GTWeave* proceeds in two phases. In the first phase, we apply the same integration procedure discussed in Section V-B above, which optimizes the plan for varying window specifications and different group-by attributes. Then, in the second phase, we augment the plan with *IPS* structures before each and every *intercede-aggregation* operator to support different predicates. Figure 9 shows such augmented plan for the ACQs of Example 4.

The detailed steps of *GTWeave* are shown in Algorithm 2 and are summarized as follows. Given a set of n ACQs, *GTWeave* first generates an optimized group-by tree that exploits the overlap between the different group-by attributes among those ACQs (as in Section V-A). Secondly, for each subset of ACQs that are rooted at a first-level node in the group-by tree, *TriWeave* is used to further generate a weaved plan that exploits the weaveability between the different windows in that subset of ACQs to selectively divide them into weaved groups (as described in Section V-B). The combination of the previous two steps results in a weaved plan that is augmented with a group-mapping table (as described in Section V-B) Finally, *GTWeave* augments the plan with the necessary *IPS* structures needed for the evaluation of the different predicates (as described in Section IV-B) and it produces the final *GTWeave* plan.

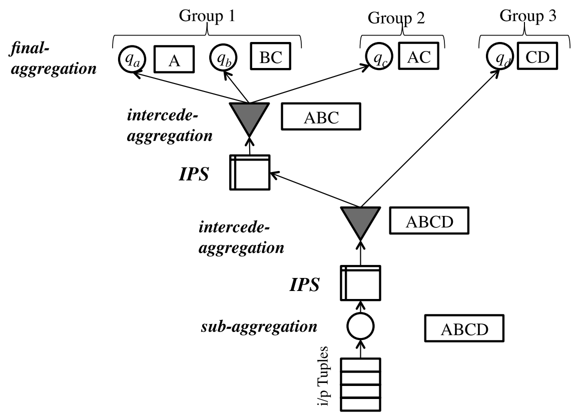


Fig. 9. *TriWeave* Plan - Varying Windows, Predicates, and Group-by

Note that *GTWeave* retains all the advantages of *TriOps* and the techniques it incorporates (such as adaptability). For example, with respect to adaptability, if the re-invocation of *GTWeave* due to addition of ACQs retains the same group-by tree but weaves the ACQs differently, then in that case only the group-mapping tables used by the intercede-aggregation operators need to be updated to reflect the new structure of the weaved groups.

VII. Performance Evaluation

It was shown experimentally that *Weave Share* outperforms the alternative sharing schemes generating up to four orders of magnitude better quality plans [12]. For this reason, in this section we focus only on showing the minimum expected benefits of *TriOps*, and consequently of *TriWeave* and *GTWeave*, by comparing *TriWeave* to *Weave Share* assuming ACQs with varying windows. The advantages of *TriOps* with *Data Shards* were discussed in Section IV-B. After describing the simulation platform in Section VII-A, we discuss the experimental results in Section VII-B.

A. Experimental Platform

We have implemented the *TriWeave* optimizer and *TriOps* in the same simulation platform previously used to evaluate *Weave Share* using *TwoOps* [12]. Below we describe the generated workload characteristics, data sets, the performance metrics and the algorithms used in our evaluations.

ACQs: We generated ACQs with different specifications (Table II). Specifically, 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) of each ACQ is set relative to its slide. That is, $r_i = \omega_i \times s_i$, where ω_i is the overlap factor, drawn from a uniform distribution.

Experimental Parameters: In each experiment, we also changed 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 than one per second) to high speed network monitoring (10K tuples/sec).

TABLE II
Simulation parameters

Parameter	Values
Slide Length	[1–10K] time units (Zipf distribution)
Slide Skewness	0.6 (moderately skewed)
Overlap Factor	[1–50] (uniform distribution)
Number of ACQs	[50–1000] queries
Input Rate	[0.5–10K] tuples/sec (Poisson distribution)

Dataset: We chose to use a synthetic workload, which allowed us to control the system parameters in order to conduct detailed sensitivity analysis while covering possible real scenarios.

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

Algorithms: We used *Weave Share* and *Shared* that use *TwoOps* (discussed in Section II) as the baseline algorithms for our comparisons. In order to get better understanding of the behavior of *TriWeave* and *TriOps*, we also evaluated different combinations of optimizers and processing models. For instance, we generated *Weave Share* and *Shared* plans that use *TwoOps*, but then ran the plan using *TriOps*.

B. Experimental Results

***TriWeave* vs *Weave Share* (Fig. 10):** In this set of experiments, we measure the performance gains of *TriWeave* by comparing the quality of the weaved plans generated by *TriWeave* and *Weave Share* for 256, 500, and 1K ACQs. We plot the normalized cost to *Weave Share* as the input rate increases. Figure 10 shows that *TriWeave* achieves up to 65% cost reduction over *Weave Share*. For low input rates and as the number of ACQs increases, the improvement is less than 40%. The reason is that for low input rates (λ), the edge rate is the dominating factor of the cost at the final-aggregation ($\sum_{i=1}^{m} E_i\Omega_i$) as well as at the intercede operator ($m.E$) (Equation 4). This also explains why at low input rate, the gain also decreases with the increase of ACQs. This is because the number of groups m increases, which increases the total cost of the plan.

***TriOps* vs. *TwoOps* (Figs. 11 – 13):** In this set of experiments, we take a closer look into the performance of *TriOps*. Specifically, we generated query plans using *Weave Share*, *Shared*, and *TriWeave*, then we measured the cost of each plan when using *TwoOps* (Equation 3) vs when using *TriOps* (Equation 4). The results are normalized to the cost of the base case, i.e., the *Shared* plan using *TwoOps*.

Figures 11, 12 and 13 show the performance gains of *TriOps* for low (50), medium (300) and high (10K tuples/second) input rates, respectively. We plot the normalized cost as the number of ACQs increases. We also highlight the trend of the *TriWeave* plan in each plot. All three figures show that for each plan, utilizing *TriOps* achieves gains over *TwoOps*, except in the case of *Shared* (which has normalized cost > 1.0). The reason is simply because when there is only one group, the *intercede-aggregation* adds an overhead with no benefit. However, when

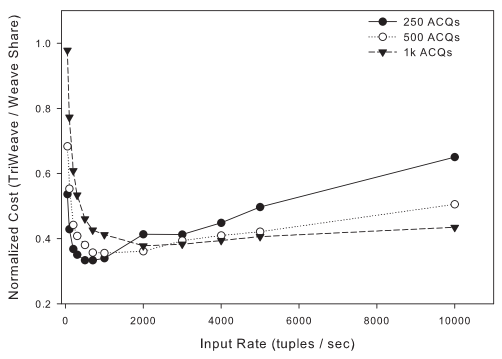


Fig. 10. *TriWeave* performance gain - Impact of Input Rate

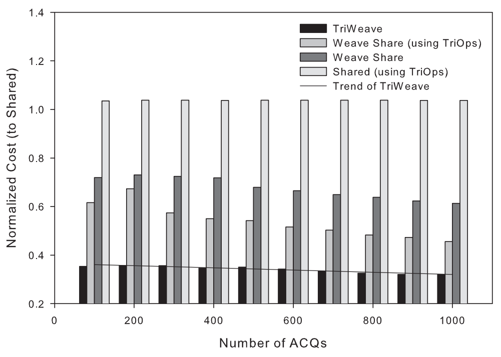


Fig. 12. Using *TriOps* processing for different plans (300 tuples/sec)

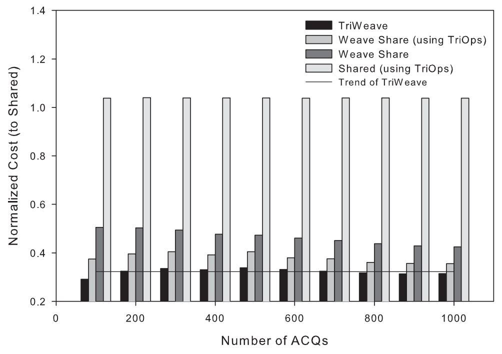


Fig. 11. Using *TriOps* processing for different plans (50 tuples/sec)

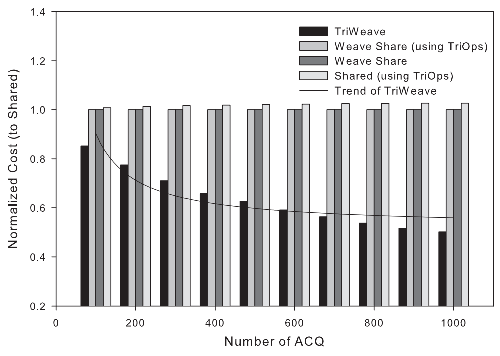


Fig. 13. Using *TriOps* processing for different plans (10K tuples/sec)

there are at least two groups, utilizing *TriOps* achieves gains between 40% and 60%.

All three figures also show that the gain under *TriOps* increases with the number of ACQs. This is mainly due to the fact that the more ACQs, the more chances for selective sharing. *TriOps*, however, allows the optimizer to take full advantage of selective sharing, whereas *TwoOps* would require duplicating the sub-aggregation operator and hence, limits the optimizer's flexibility in utilizing selective sharing. We have confirmed this intuition by checking the number of weaved groups each scheme has produced, and finding that *TriOps* consistently leads to plans with much larger number of groups.

Finally, the rate with which the gain of *TriOps* increases as the number of ACQs increases is faster for higher input rates. The reason is that at higher input rates, *TriOps* achieves larger reductions by replacing the multiple sub-aggregation operators by one sub-aggregation and one *intercede-aggregation*.

VIII. Related Work

Utilizing partial aggregation is the underlying principle for the optimized processing of individual ACQs. The *Pane* [17] scheme splits each slide into smaller fragments to be processed by the sub-aggregation operator. While *Pane* splits the window into multiple equal-sized fragments, *Paired Window* [16] improves over *Pane* by splitting each slide into at most two fragments to minimize the processing needed at the final-aggregation operator.

The *Window-ID* (WID) technique proposed in [18] improves the performance of an ACQ by maintaining multiple aggregates for multiple window extents at the same time. A bucket operator is then utilized to tag each input tuple with the range of those window extents that this tuple belongs to and to aggregate it into all its window extents at once.

The above schemes use the same underlying *TwoOps* implementation of the aggregate operator. There are also different models for processing ACQs. For example, optimization techniques for processing sliding-window queries (including ACQs) that utilize the *negative tuples* approach have been proposed in [9]. In the negative tuples approach, tuple expiration is determined when a negative tuple is inserted.

In general, there is a rich literature on multiple query optimization (MQO) in traditional databases [23], [22], [20], [14], [28], as well as in data streams [28], [2], [29], [19]. MQO in traditional databases aims at exploiting common sub-expressions to reduce evaluation cost. Finding the optimal query plan, in traditional databases, as well as in data streams, is an NP-Hard problem, and hence heuristic approaches were investigated. For instance, two cost-based and one greedy heuristics were proposed in [22]. The two cost-based heuristics extend the *Volcano* [11] query optimizer by performing a depth-first search in the state space of alternative query plans, to detect common sub-expressions across different queries.

In [30], [31], [21], the problem of optimizing multiple ACQs with different group-by attributes was addressed along the lines of classical subsumption-based multi-query optimization techniques. The proposed *Phantoms* technique [30], [31] essentially introduces a set of sub-aggregates (i.e., phantoms) to share the processing of ACQs with similar windows and predicates but different group-by attributes, each of which can be derived from the *Phantom*. In [21], the *Intermediate-aggregates* techniques generalizes *Phantoms* and uses a greedy heuristic approach to generate a group-by tree with minimal cost, subject to a memory constraint.

The introduction of window-based continuous queries for the processing of statefull operators (i.e., joins and aggregates) over unbounded data streams has motivated the recent work discussed earlier on *Shared Time Slices* and *Shared Data Shards* [16], including ours [12], on the shared processing of queries with overlapping windows.

At the processing level, sharing the results of aggregation among different ACQs has also been proposed in [10], where a scheduling technique to optimize the execution of ACQs has been developed. This technique utilizes a window-aware scheduling scheme that synchronizes the re-execution times of similar CQs to execute common parts only once. This is to be distinguished from *TriWeave* which shares the sub-aggregation operator, regardless of the scheduling policy.

IX. Conclusions

All current multiple ACQs optimization techniques exploit the concept of partial aggregation to minimize the repeated processing of overlapping windows. In this paper, we questioned the effectiveness of the widely accepted two-level or two-operator implementation of partial aggregation (*TwoOps*) and proposed to replace it with *TriOps*, a new three-level processing model. In particular, we introduced in *TriOps* the *intercede-aggregation* operator between the sub-aggregation and final-aggregation operators. We showed the effectiveness of *TriOps* in the context of the Weavability-based optimizers which selectively group ACQs with varying windows into multiple weaved groups (query execution trees).

Specifically, we illustrated that the proposed *intercede-aggregation* operator in *TriOps* minimizes the total cost of processing multiple ACQs by allowing sharing of the sub-aggregation across all weaved groups, and supports the shared processing of multiple ACQs with different predicates or group-by attributes. Further, we developed *TriWeave*, a *TriOps*-aware multiple ACQs optimizer along the lines of the Weave Share optimizer and proposed *GTWeave* that integrates *TriWeave* and the *Intermediate-aggregates* optimizer to support multiple ACQs with different window specifications, different predicates and different group-by attributes. Our simulation results demonstrated the applicability and performance benefits of *TriOps* and consequently of *TriWeave* and *GTWeave*.

Acknowledgments This work was supported in part by National Science Foundation awards IIS-0534531, IIS-0746696, and IIS-1050301, an Andrew Mellon Predoctoral Fellowship, and Australian Research Council award DP110102777. Finally, we would like to thank the anonymous reviewers for their helpful and insightful comments.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. C. etintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.
[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 2003.
[3] Aqsios, http://db.cs.pitt.edu/group/projects/aqsios, 2011.
[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD*, 2003.
[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, 2003.
[6] P. K. Chrysanthis. Aqsios - next generation data stream management system. *CONET Newsletter*, June 2010.
[7] Coral8, http://www.coral8.com/, 2004.
[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.
[9] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE TKDE*, 2007.
[10] L. Golab, K. G. Bijay, and M. T. Ozsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, 2006.
[11] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
[12] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.
[13] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Y. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, 2004.
[14] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.
[15] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, 2010.
[16] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
[17] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 2005.
[18] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.
[19] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
[20] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, 2001.
[21] K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.
[22] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
[23] T. K. Sellis. Multiple-query optimization. *ACM TODS.*, 13(1), 1988.
[24] M. Sharaf, P. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM TODS*, 33, 2008.
[25] Streambase, http://www.streambase.com, 2006.
[26] System S, http://domino.research.ibm.com/, 2008.
[27] Truviso, http://www.truviso.com, 2005.
[28] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.
[29] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: new paradigm of multi-query optimization of window-based stream queries. In *VLDB*, 2006.
[30] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.
[31] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *VLDB Journal*, 2010.