Processing of Aggregate Continuous Queries in a Distributed Environment

Anatoli U. Shein, Panos K. Chrysanthis, Alexandros Labrinidis

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260
Email: {aus, panos, labrinid}@cs.pitt.edu

Abstract. Data Stream Management Systems (*DSMSs*) performing online analytics rely on the efficient execution of large numbers of Aggregate Continuous Queries (*ACQs*). In this paper, we study the problem of generating high quality execution plans of *ACQs* in *DSMSs* deployed on multi-node (multi-core and multi-processor) distributed environments. Towards this goal, we classify optimizers based on how they partition the workload among computing nodes and on their usage of the concept of *Weavability*, which is utilized by the state-of-the-art *WeaveShare* optimizer to selectively combine *ACQs* and produce low cost execution plans for single-node environments. For each category, we propose an optimizer, which either adopts an existing strategy or develops a new one for assigning and grouping *ACQs* to computing nodes. We implement and experimentally compare all of our proposed optimizers in terms of (1) keeping the total cost of the *ACQs* execution plan low and (2) balancing the load among the computing nodes. Our extensive experimental evaluation shows that our newly developed *Weave-Group to Nodes* (WG_{TN}) and *Weave-Group Inserted* (WG_I) optimizers produce plans of significantly higher quality than the rest of the optimizers. WG_{TN} minimizes the total cost, making it more suitable from a client perspective, and WG_I achieves load balancing, making it more suitable from a system perspective.

1 Introduction

Nowadays, more and more applications are becoming available to wider audiences, resulting in an increasing amount of data being produced. A large volume of this generated data often takes the form of high velocity streams. At the same time, online data analytics have gained momentum in many applications that need to ingest data fast and apply some form of computation, such as predicting outcomes and trends for timely decision making.

In order to meet the near-real-time requirements of these applications, Data Stream Management Systems (*DSMS*) [4, 24, 5, 25, 18] have been developed to efficiently process large amounts of data arriving with high velocities in the form of streams. In *DSMSs*, clients register their analytics queries, which consist of one or more *Aggregate Continuous Queries* (*ACQs*). *ACQs* continuously aggregate streaming data and periodically produce results such as *max*, *count*, *sum*, and *average*.

A representative example of online analytics can be found in stock market web applications where multiple clients monitor price fluctuations of stocks. In this setting, a

system needs to be able to answer analytical queries (i.e., average stock revenue or profit margin per stock) for different clients, each one with potentially different relaxation requirements in terms of accuracy.

Another example is the monitoring of personal health data (e.g., heart beats per minute) and physical activity data (e.g., number of steps walked, number of miles ran) of individuals, along with location and environmental data (e.g., barometric pressure), generated by devices such as Fitbit, Apple iWatch, etc. This data serves as input to a set of monitoring applications, which are implemented as *ACQs* that may execute for a user or on behalf of a user such as by the user's primary care physician or health insurance companies to which the user may have only allowed aggregate-level queries.

The accuracy of an *ACQ* can be thought of as the window in which the aggregation takes place and the period at which the answer is recalculated. Periodic properties that are used to describe *ACQs* are *range* (**r**) and *slide* (**s**) (sometimes also referred to as *window* and *shift* [13], respectively). A slide denotes the interval at which an *ACQ* updates its result; a range is the time window for which the statistics are calculated. For example, if a stock monitoring application has a slide of 3 sec and a range of 5 sec, it means that the application needs an updated result every 3 sec, and the result should be derived from data accumulated over the past 5 sec.

DSMSs are required to maintain *ACQs*' state over time, while performing aggregations. *ACQs* with a larger range will have a higher cost to maintain its state (memory) and compute its results (CPU). The most space and time efficient method to compute aggregations is to run partial aggregations on the data while accumulating it, and then produce the answer by performing the final aggregation over the partial results (Sec. 2).

In order to cope with the sheer volume of information, enterprises move to distributed processing infrastructures such as local clusters or the *Cloud*. The deployment of *DSMSs* to *Cloud* results in multi-tenant settings, where multiple *ACQs* with even more diverse periodic properties are executed on the same hardware.

Problem Statement It is safe to say that the efficiency of *DSMSs* deployed on multiple multi-core computing nodes depends on the *intelligent* collocation of *ACQs* operating on the same data streams and calculating similar aggregate operations. If such *ACQs* have similarities in their periodic properties, the opportunity to share final and partial results arises, which can reduce the overall processing costs.

Typically, the number of *ACQs* with similar aggregation types for a given data stream can be overwhelming in online systems [5]. Therefore, it is crucial for the system to be able to make decisions quickly on combining different *ACQs* in such a way that would benefit the system. Unfortunately, this has been proven to be NP-hard [27], and, currently, only approximation algorithms can produce acceptable execution plans. For instance, the state-of-the-art *WeaveShare* optimizer [12], which selectively combines *ACQs* and produces high quality plans, is theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem [8].

Under these circumstances, *it is vital to develop efficient data sharing schemes among* ACQs *that lead to an effective assignment of* ACQs *to computing nodes*.

Our Approach The state-of-the-art *WeaveShare* optimizer is a cost-based *ACQ* optimizer that produces low cost execution plans by utilizing the concept of *Weavability* [12]. Since *WeaveShare* is targeting single-node *DSMSs*, it is oblivious to distributed

processing capabilities, and as our experiments have revealed, *WeaveShare* cannot produce *ACQ* execution plans of equivalent cost that can be assigned to the various computing nodes. This motivated us to address the problem of generating high quality execution plans of *ACQs* in *DSMSs* deployed on multi-node (multi-core and multi-processor) distributed environments with a *Weavability*-based optimizer.

Formally, given a set $\mathscr{Q}$ of all *ACQs* submitted by all clients and a set $\mathscr{N}$ of all available computing nodes in the distributed *DSMS*, our goal is to find an execution plan $\mathscr{P}(\mathscr{Q}, \mathscr{N}, \mathscr{T})$ that maps $\mathscr{Q}$ to $\mathscr{N}$ ($\mathscr{Q} \rightarrow \mathscr{N}$) and generates a set $\mathscr{T}$ of local *ACQ* execution trees per node, such that the total cost of the *ACQs* execution is low and the load among the computing nodes is balanced.

The rationale behind these two optimization criteria is (Sec. 3):

- *Minimizing the total cost* of the execution plan allows the system to support more *ACQs*. In the case of the *Cloud*, since *Cloud* providers charge money for the computation resources, satisfying more client requests using the same resources results in less costly client requests.
- *Balancing the workload* among computation nodes saves energy while still meeting the requirements of the installed *ACQs*, which directly translates to monetary savings for the distributed infrastructure providers. Additionally, it is advantageous for the providers to maintain load balancing, because it prevents the need to overprovision in order to cope with unbalanced workloads.

Contributions We make the following contributions:

- We explore the challenges of producing high quality execution plans for distributed processing environments and categorize possible *ACQ* optimizers for these environments based on how they utilize the concept of Weavability for cost-based optimization as shown in Table 1. (Sec. 4)
- We propose an *ACQ* optimizer for each category. These optimizers either adopt an existing strategy or develop a new one for assigning and grouping *ACQs* to computing nodes. (Secs. 5 and 6)
- We experimentally evaluate our optimizers and show that our newly developed *Weave-Group to Nodes* optimizer is the most effective in terms of minimizing the total cost of the execution plan, making it more suitable from the clients' perspective, and our *Weave-Group Inserted* optimizer is the most effective in terms of achieving load balance, making it more suitable from a system perspective. Both produce quality plans that are orders of magnitude better than the other optimizers. (Sec. 7)

2 Background

In this section we briefly review the underlying concepts of our work, namely *partial aggregation* and *Weavebility*.

Partial aggregation was proposed to improve the processing of *ACQs* [10, 15–17]. The idea behind partial aggregation is to calculate partial aggregates over a number of partitions, then assemble the final answer by performing the final aggregation over these aggregates. As opposed to partial aggregation in traditional database systems where

partitioning is value-based, partial aggregation in *DSMSs* uses time-based (or tuple-based) partitioning.

Partial aggregations, as shown in Fig. 1, are implemented as *two-level operator trees*, consisting of the partial- or sub-aggregator and the final-aggregator. The *Paired Window* technique, [15], also shown in Fig. 1, is the most efficient implementation of partial aggregations. This technique does not assume any relation between range and slide and uses two fragment lengths, g_1 and g_2, where $g_1 = range\%slide$ and $g_2 = slide - g_2$. Partial aggregations are computed at periods of fragment g_1 and fragment g_2 interchangeably.

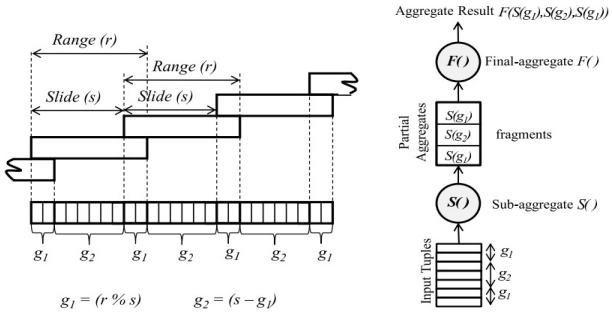

Fig. 1: Paired Window Technique

Shared Processing of ACQs Several processing schemes, as well as multiple *ACQ* optimizers, utilize the *Paired Windows* technique [15, 12]. To show the benefits of sharing partial aggregations, consider the following example:

Example 1 Assume two *ACQs* that perform *count* on the same data stream. The first *ACQ* has a slide of 2 sec and a range of 6 sec, the second one has a slide of 4 sec and a range of 8 sec. That is, the first *ACQ* is computing partial aggregates every 2 sec, and the second is computing the same partial aggregates every 4 sec. Clearly, the calculation producing partial aggregates only needs to be performed once every 2 sec, and both *ACQs* can use these partial aggregates for their corresponding final aggregations. The first *ACQ* will then run each final aggregation over the last three partial aggregates, and the second *ACQ* will run each final aggregation over the last 4 partial aggregates.

To determine how many partial aggregations are needed after combining *n ACQs*, we need to first find the length of the new combined (composite) slide, which is the *Least Common Multiple* (*LCM*) of all the slides of combined *ACQs*. Each slide is then repeated $LCM/slide$ times to fit the length of the new composite slide. All partial aggregations happening within each slide are also repeated and marked in the composite slide as *edges* (to mark the times at which partial aggregations will be happening). If two *ACQs* mark the same location, it means that location is a *common edge*.

To count how many partial aggregations (*edges*) are scheduled within the composite slide we can use either the *Bit Set* technique [12] or the *Formula F1* technique [23].

Weavability [12] is a metric that measures the benefit of sharing partial aggregations between any number of *ACQs*. If it is beneficial to share computations between these *ACQs*, then these *ACQs* are known to *weave* well together and are combined into the same shared execution tree. Intuitively, two *ACQs weave* perfectly when their *LCM* contains only *common edges*.

The following formula can be used to calculate the cost (C) of the execution plan before and after combining *ACQs* from their own trees into shared trees. The difference between these costs tells us if the combination is a good choice.

$$C = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{1}$$

Where m is the number of the trees in the plan, λ is input rate in tuples per second, E_i is *Edge rate* of tree i, and Ω_i is the overlap factor of tree i. *Edge rate* is the number of partial aggregations performed per second, and the overlap factor is the total number of final-aggregation operations performed on each fragment.

The WeaveShare optimizer utilizes the concept of *Weavability* to produce an execution plan for a number of *ACQs*. It selectively partitions the *ACQs* into multiple disjoint execution trees (i.e., groups), resulting in a dramatic reduction in total query plan processing cost. *WeaveShare* starts with a no-share plan, where each *ACQ* has its own execution tree. Then, it iteratively considers all possible pairs of execution trees and combines those that reduce the total plan cost the most into a single tree. *WeaveShare* produces a final execution plan consisting of multiple disjoint execution trees when it cannot find a pair that would reduce the total plan cost further.

3 System Model and Execution Plan Quality

In this paper, we assume a typical *DSMS* deployed over a set of servers (i.e., computing nodes). These servers can be a local cluster or on the *Cloud* and are capable of executing any *ACQs* using partial aggregation. Submitted *ACQs* are assumed to be independent of each other and have no affinity to any server. Furthermore, without a loss of generality, we target *ACQs* that perform similar aggregations on the same data stream.

In a single node system, the main metric defining the quality of an execution plan is the *Cost* of the plan. The *Cost* of the plan is measured in operations per second. That is, if the plan cost is X, then we would need a server that can perform at least X operations per second in order to execute this plan and satisfy all users by returning the results of their *ACQs* according to their specified range and slide.

In the context of the distributed environment, we have to split our workload between the available nodes. Since our workload consists of *ACQs*, we can assign them to the available computing nodes in the system and group them into execution trees within these nodes. Thus, in any distributed environment, the *Total Cost* of a plan P is calculated as a sum of all costs C_i (according to the Eq. 1) of all n nodes in the system:

$$TotalCost(P) = \sum_{i=1}^{n} C_i$$

Table 1: Optimizer Categories

		Optimizers				
		Non-Cost-based		Cost-based		
		Random	Round Robin	to Lowest	to Nodes	inserted
Categories	Group Only	G_{RAND}	G_{RR}	G_{TL}	-	-
	Weave Only	W_{RAND}	W_{RR}	W_{TL}	W_{TN}	W_I
	Weave + Group	WG_{RAND}	WG_{RR}	WG_{TL}	WG_{TN}	WG_I

This metric is important for the *Cloud* environment, because lowering the total cost T allows *DSMSs* to handle larger numbers of different *ACQs* on the same hardware, which in turn can potentially lower the monetary cost of each *ACQ* for the clients.

Another important metric in a distributed environment is the *Maximum node cost* of all computational nodes. The maximum node cost of a plan P is calculated by finding the highest cost C_i of all n nodes in the system:

$$MaxCost(P) = Max_i^n C_i$$

Minimizing the *Max Cost* is vital for distributed *DSMSs* with heavy workloads. In such a case, if we optimize our execution plans purely for the *Total Cost*, due to the heavy workload, the *Max Cost* can become higher than the computational capacity of the highest capacity node in the system, and the system will not be able to accommodate this execution plan. Furthermore, it is advantageous for the providers to maintain load balancing, because it prevents the need for over-provisioning in order to cope with unbalanced workloads.

Additionally, good load balancing could enable power management that executes *ACQs* at lower CPU frequency. This could lead to significant energy savings, ergo monetary savings, given that the energy consumption is at least a quadratic function of CPU frequency [26].

4 Taxonomy of Optimizers

As mentioned in the Introduction, in order to structure our search for a suitable multi-*ACQ* optimizer for a distributed *DSMS* in a systematic way, we categorize possible *ACQ* optimizers based on how they utilize the concept of *Weavability* for both non-cost-based and cost-based optimization. This taxonomy is shown in Table 1. Below, we highlight the underlying strategy of each category.

Group Only This category allows for the grouping of *ACQs* on different computation nodes. No sharing of final or partial aggregations between *ACQs* is allowed. Optimizers in this category are expected to be effective in environments where sharing partial aggregates is counter productive, for example, when there are no similarities between periodic properties of *ACQs*. Even though there is no sharing between *ACQs* in this category, it is still essential to maintain the load balance between computation nodes in a distributed environment. Since node costs in this case are calculated trivially by adding

together separate costs of *ACQs* running on this node, there can be many analogies (such as CPU scheduling in OS) to optimizers from this category.

Weave Only This category allows the sharing of final and partial aggregations between *ACQs*. The *Weavability* concept is used in this category to generate the number of execution trees matching the number of available nodes. As a result, only one execution tree can be present on each computation node in the resulting plan. Optimizers in this category are expected to be effective in the environments where partial result sharing is highly advantageous, for example, if the submitted *ACQs* all have similar periodic properties (*ACQ* slides are the same or multiples of each other).

Weave and Group This category allows both the sharing of aggregations between *ACQs* within execution trees and the grouping of them on different computation nodes. Thus, multiple execution trees can be present on any node. Optimizers in this category are attempting to be adaptive to any environment and produce high quality execution plans in different settings by collocating and grouping *ACQs* in an intelligent way.

5 Non-Cost-based Optimizers

In this section, we provide the details on the *Non-Cost-based* optimizers, which we further classified as *Random* and *Round Robin* optimizers. Random and Round Robin optimizers iterate through a set of input *ACQs*, selecting a node for each *ACQ* in a random or round robin fashion respectively.

Depending on the way *ACQs* on a node are woven,

- G_{RAND} & G_{RR} (*GroupOnly*) add the *ACQs* to the selected node as a separate tree.
- W_{RAND} & W_{RR} (*WeaveOnly*) *weave* the *ACQs* into a single, shared tree on the node.
- WG_{RAND} & WG_{RR} (*WeaveandGroup*) choose (in random or round robin fashion) whether to add this *ACQ* as a separate tree, or to *weave* it with one of the available trees on this node.

6 Cost-based Optimizers

In this section, we provide the details on the second class of optimizers: *Cost-based* optimizers (Table 1), which includes three categories: “To Lowest”, “To Nodes”, and “Inserted”. Note that no representatives for the “Group Only – Insert" and “Group Only – To Nodes" categories are listed in Table 1 because in both cases the representative is effectively G_{TL} without weaving. In all optimizers, we consider the initial cost of each node to be zero.

6.1 Category “To Lowest”

Optimizers in this category follow the “To Lowest” algorithm shown in Algorithm 1.

Group to Lowest (G_{TL}) This optimizer is a balanced version of a *No Share* generator, which assigns each *ACQ* to run as a separate tree and that are then assigned to available nodes in a cost-balanced fashion.

Algorithm 1 The *"To Lowest"* Algorithm

```
Input: A set of Q Aggregate Continuous Queries, N computation nodes, and Category
Output: Execution plan P
Create an execution tree (t_1, t_2, ..., t_Q) for each query
Calculate costs for all execution trees (c_1, c_2, ..., c_Q)
Sort all execution trees from expensive to cheap
Assign N most expensive trees to N nodes (n_1, n_2, ..., n_N)        ▷ assign one tree per node
T ← Q − N                          ▷ T is the number of remaining trees to be grouped/weaved
for i = 0 to T do                  ▷ iterate over the trees until all are grouped/weaved to nodes
    MinNode ← findMinNode()        ▷ determine the node with the current smallest cost
    switch Category do
        case GroupOnly                                   ▷ each node can have multiple trees
            group(t_i, MinNode)                          ▷ group t_i as a separate tree to MinNode
        case WeaveOnly                                   ▷ each node can have only one tree
            weave(t_i, MinNode)                          ▷ weave t_i to the tree in MinNode
        case WeaveAndGroup                               ▷ each node can have multiple trees
            Cost_1 ← group(t_i, MinNode)   ▷ new cost of MinNode if t_i is grouped to MinNode
            MinTree ← findMinTree(MinNode)               ▷ minimal costing tree in MinNode
            Cost_2 ← weave(t_i, MinNode)   ▷ new cost of MinNode if t_i is weaved to MinTree
            if Cost_1 < Cost_2 then
                group(t_i, MinNode)                      ▷ group t_i as a separate tree to MinNode
            else
                weave(t_i, MinNode)                      ▷ weave t_i to MinTree
            end if
    end switch
end for
end (Return P)
```

Algorithm: The trees are first sorted by their costs, then, starting from the most expensive one, each tree is assigned to the node that currently has the lowest total cost.
Discussion: Since this optimizer does not perform any partial result sharing, it is only useful in cases when sharing is not beneficial (when none of the slides have any similarities in their periodic features).

Weave To Lowest (W_{TL}) This optimizer builds on the G_{TL} algorithm and weaves all *ACQs* on a node into a single, shared tree.
Algorithm: After sorting *ACQs* by cost (as in G_{TL}), W_{TL} assigns each *ACQ* to a node with the current lowest total cost and weaves it into the shared tree on the node.
Discussion: The W_{TL} optimizer executes *Weavability* calculation only once per input which makes it more expensive to run than G_{TL}. Additionally, by limiting to a single shared tree and not considering the compatibility of existing *ACQs* with new ones, it produces plans with high *Total Cost*, and, consequently, high *Max Cost*, even though it performs rudimentary cost balancing.

Weave-Group To Lowest (WG_{TL}) This approach also builds on G_{TL}, but as opposed to W_{TL}, it allows both *selective weaving* and grouping *ACQs* together.

Algorithm: Similar to G_{TL} and W_{TL}, WG_{TL} first sorts the *ACQ* trees, then iteratively assigns each *ACQ* to the node with the current smallest cost. At a node, an *ACQ* is either *woven* with the smallest costing tree in the node or added as a separate tree, whichever leads to the minimum cost increase.
Discussion: The WG_{TL} has similar runtime cost as W_{TL} as both optimizers use the *Weavability* calculations only once per *ACQ*. Even though WG_{TL} attempts to take advantage of grouping, it does not produce much better execution plans than W_{TL}. By focusing only on the lowest cost tree on a node, it weaves together some poorly compatible *ACQs*, leading to comparatively low quality execution plans.

6.2 Category "To Nodes"

Optimizers in this category follow the "To Nodes" algorithm depicted in Algorithm 2.

Weave to Nodes (W_{TN}) This optimizer is directly based on the single node *WeaveShare* algorithms, thus it is targeted at minimizing the *Total Cost*.
Algorithm: W_{TN} starts its execution the same way as the single node *WeaveShare*. If it reaches the point where the current number of trees is less than or equal to the number of available nodes, W_{TN} stops and assigns each tree to a different node. If, however, *WeaveShare* finishes execution, and the current number of trees is still greater than the number of available nodes, the W_{TN} optimizer continues the *WeaveShare* algorithm (merging trees pairwise), even though it is no longer beneficial for total cost. The execution stops when the number of trees becomes equal to the number of available nodes.
Discussion: Since W_{TN} is a direct descendant of *WeaveShare*, it is optimized to produce the minimum *Total Cost*. However, since W_{TN} allows only one execution tree per node, in order to match the number of nodes to number of trees, W_{TN} forces *WeaveShare* to keep merging trees with less compatible *ACQs*. Hence, W_{TN} generates, in general, more expensive plans than the basic *WeaveShare*. Additionally, W_{TN} does not perform any load balancing, hence it can generate query plans with execution trees whose computational requirements exceed the capacity of the node with the most powerful CPU.

Weave-Group to Nodes (WG_{TN}) Like W_{TN}, this optimizer is also directly based on the single node *WeaveShare* algorithm and is targeted at minimizing the *Total Cost*.
Algorithm: The WG_{TN} optimizer starts by executing single core *WeaveShare* and, similarly to W_{TN}, stops execution if it reaches the point where the current number of trees is equal to or less than the number of available nodes. However, if *WeaveShare* finishes execution and the current number of trees produced is greater than the number of available nodes, WG_{TN} assigns them to the available nodes, without *weaving* them, in a balanced fashion by applying the G_{TL} optimizer. First, all trees are sorted by their costs, and, starting from the most expensive ones, the trees are assigned to the nodes with the smallest current total cost.
Discussion: Unlike W_{TN}, the WG_{TN} optimizer is designed to produce the minimum *Total Cost* and the minimum *Max Cost*. The latter is not always possible, since the execution trees produced by *WeaveShare* are sometimes of significantly different costs, and the used load balancing technique cannot produce the desired output. WG_{TN} can achieve a better *Total Cost* than W_{TN} by not forcing trees that do not *weave* well together to merge, which would have increased the total cost of the plan. However, the penalty

Algorithm 2 The *"To Nodes"* Algorithm

```
Input: A set of Q Aggregate Continuous Queries, N computation nodes, and Category
Output: Execution plan P
Create an execution tree (t_1, t_2, ..., t_Q) for each query
T ← Q                                          ▷ T is the number of remaining trees
loop
    MaxReduction ← −∞                          ▷ maximum cost reduction is set to minimum
    for i = 0 to T − 1 do                      ▷ iterate over all trees
        for j = 1 to T do                      ▷ iterate over all trees again (to cover all pairs)
            CostRed ← weave(t_i, t_j)          ▷ cost reduction if weaving trees t_i and t_j
            if CostRed > MaxReduction then     ▷ find largest CostRed
                MaxReduction ← CostRed         ▷ and save it to MaxReduction
                ToWeave ← (t_i, t_j)           ▷ trees t_i and t_j are saved to be weaved later
            end if
        end for
    end for
    if MaxReduction > 0 then                   ▷ there is a benefit in weaving
        weave(ToWeave)                         ▷ weave saved trees
    else
        switch Category do
            case WeaveOnly
                if T ≤ N then
                    end (Return P)
                else
                    weave(ToWeave)             ▷ weave saved trees
                end if
            case WeaveAndGroup
                P ← G_TL(T)                    ▷ run GroupToLowest optimizer on remaining T trees
                end (Return P)
        end switch
    end if
    T ← T − 1
end loop
```

of grouping execution trees on nodes without merging them is that each tuple has to be processed as many times as the number of trees on a node. This effectively increases the *Total Cost* by a factor equal to the input rate multiplied by the number of the trees on each node. Clearly, the higher the input rate of a stream, the more costly it will be for the system to group trees without *weaving* them.

6.3 Category "Inserted"

Optimizers in this category follow the "Inserted" algorithm depicted in Algorithm 3.

Weave Inserted (W_I) This approach is based on the *Insert-then-Weave* optimizer introduced in [12], in which every *ACQ* is either weaved in an existing tree or assigned to a new tree, whichever results in the smallest increase in the *Total Cost*. The difference of

Algorithm 3 The *"Inserted"* Algorithm

```
Input: A set of Q Aggregate Continuous Queries, N computation nodes, and Category
Output: Execution plan P
Assigning first N queries to N nodes (n_1, n_2, ..., n_N) as separate trees
Calculate node costs for all N nodes
Q ← Q − N                                  ▷ Q is the number of remaining queries to be assigned
WeaveCost ← ∞                              ▷ weave cost is set to maximum
for i = 0 to Q do                          ▷ iterate over the queries until all are grouped/weaved
    MinNode ← findMinNode()                ▷ determine the node with the current smallest cost
    for j = 0 to N do                      ▷ iterate over all nodes
        for k = 0 to NumTrees in n_j do    ▷ iterate over all trees within a node
            TempCost ← weave(q_i, t_k)     ▷ determine plan cost if weaving query q_i into tree t_k
            if TempCost < WeaveCost then   ▷ find smallest TempCost
                WeaveCost ← TempCost       ▷ and save it to WeaveCost
                ToWeave ← (q_i, t_k)       ▷ query q_i is saved to be weaved to tree t_j later
            end if
            switch Category do
                case WeaveOnly
                    weave(ToWeave)         ▷ weave saved trees
                case WeaveAndGroup
                    GroupCost ← group(q_i, MinNode)   ▷ cost of MinNode if q_i is grouped
                    if GroupCost < WeaveCost then
                        group(q_i, MinNode)           ▷ group q_i to MinNode as a separate tree
                    else
                        weave(ToWeave)     ▷ weave saved trees
                    end if
            end switch
        end for
    end for
end for
end (Return P)
```

the W_I optimizer from the original *Insert-then-Weave* approach is that W_I keeps a fixed number of trees equal to the number of nodes in the distributed system.

Algorithm: W_I starts by randomly assigning an *ACQ* to each available node, then iterating through the remaining *ACQs*. For each node it computes the new cost if the *ACQ* under consideration is woven into the execution tree on the node and assigns the *ACQ* to the node that has the smallest new cost.

Discussion: W_I is attempting to optimize for the *Max Cost*, as well as the *Total Cost*, by taking into account both the *Weavability* of the inserted *ACQ* with every available node and performing cost-balancing of the computation nodes. The downside of W_I is that, since load balancing is the first priority of W_I, it sometimes assigns *ACQs* to nodes with underlying trees with which they do not *weave* well. This happens in cases where the tree that *weaves* poorly with the incoming *ACQ* currently has the smallest cost. Additionally, since W_I is limited to one execution tree per node, the *ACQs* that do not

weave well with any of the available trees are still merged into one of these trees. This increases the *Total Costs* of the generated plans.

Weave-Group Inserted (WG_I) This optimizer is also a version of the *Insert-then-Weave* approach and similar to W_I. However, since the WG_I optimizer does not have to be limited to only one execution tree per node, it utilizes grouping to keep the *Total Cost* low while maintaining load balance between nodes.
Algorithm: WG_I starts by randomly assigning an *ACQ* to each available node, then iterating through the remaining *ACQs* similarly to W_I. By trying to weave each *ACQ* under consideration into every execution tree in every node, WG_I determines each node's minimum new cost and the most compatible underlying tree. Finally, the *ACQ* is either woven to the selected tree on the node with the minimum new cost or added as a separate tree to the tree with the minimum old cost, based on which option leads to the minimum *Total Cost* increase.
Discussion: WG_I is optimized for both *Max Cost* and *Total Cost*. However, even though WG_I allows grouping of execution trees, it does not always achieve a good *Total Cost*. This happens (similarly to W_I) in cases when the tree that *weaves* poorly with the *ACQ* under consideration has the smallest cost and is located in the node with the smallest current node cost, which forces WG_I to *weave* the non-compatible *ACQs*.

Note A preprocessing step can be carried out for all optimizers by merging all *ACQs* with identical slides into the same trees, since such *ACQs weave* together perfectly. This reduces the workload down to a number of execution trees with multiple *ACQs* with the same slides. Note that this preprocessing is always beneficial in terms of the *Total Cost*, however, it is only beneficial in terms of the *Max Cost* if the distributed system has low number of nodes compared to the number of input *ACQs*. Otherwise, since the number of entities in the workload is decreased, it is more challenging to achieve balance among the high number of computating nodes.

7 Experimental Evaluation

In this section, we summarize the results of our experimental evaluation of all the optimizers for distributed processing environments listed in Table 1.

7.1 Experimental Testbed

In order to evaluate the quality of our proposed optimizers, we built an experimental platform and implemented all of the optimizers discussed above using Java. Our **workload** is composed of a number of *ACQs* with different characteristics. We are generating our workload synthetically in order to be able to fine-tune system parameters and perform a more detailed sensitivity analysis of our optimizers' performance. Moreover, it allows us to target many possible real-life scenarios and analyze them.
The **simulation parameters** utilized in our evaluation are:

- Number of *ACQs* (Q_{num}) that are installed on the same data stream and can share partial aggregations.
- Number of nodes in the target system (N_{num}).

- The input rate (λ), which describes how fast tuples arrive through the input stream.
- Maximum slide length (S_{max}), which provides an upper bound on the length of the slides of our *ACQs*. The minimum slide length allowed by the system equals one.
- Zipf distribution skew (Z_{skew}), which depicts the popularity of each slide length in the final set of *ACQs*. Zipf skew of zero produces uniform distribution, and Zipf skew of 1 is skewed towards large slides (for a more realistic example).
- Maximum overlap factor (Ω_{max}), which defines the upper bound for the overlap factor. The overlap factor of each *ACQ* is drawn from a uniform distribution between one and the maximum overlap factor.
- Generator type (*Gen*), which defines whether the workload is normal (*Nrm*), which includes any slides or diverse (*Div*), which includes only slides of a length that is a prime number. When the slides are prime, their *LCM* is equal to their product, which makes it more difficult to share partial aggregations.

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. Thus, our comparison does not include the actual execution of the plans on a distributed environment, which is part of our future work. All results are taken as averages of running each test three times. We ran all of our experiments on a single-node dual processor 8 core Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz server with 96 GB of RAM available.

7.2 Experimental Results

Experiment 1: Comprehensive Evaluation of Distributed Environment Optimizers

Configuration (Table 2) To compare the quality of produced plans by the distributed optimizers, we tried to cover as broad a range of different parameters as possible. Thus, we ran a set of 256 experiments, which correspond to all possible combinations of the parameters from Table 2 (i.e., our entire search space). For each one of these experiments, we generated a new workload according to the current parameters and executed all of the above mentioned optimizers on this workload.

Results (Fig. 2 and Tables 3 and 4). Out of a very large number of results, we observed that the *Weave to Nodes* (W_{TN}) and *Weave-Group to Nodes* (WG_{TN}) produced good plans in terms of *Total Cost*, while *Weave Inserted* (W_I) and *Weave-Group Inserted* (WG_I) performed the best in terms of *Max Cost* (Fig. 2). However, we noticed that in the majority of the cases where the W_{TN} and W_I optimizers produced the best plans (in terms of *Total Cost* and *Max Cost*, respectively), their matching optimizers from the *Weave and Group* category (WG_{TN} and WG_I) produced output of either equal or very

Table 2: Experimental Parameter Values (Total number of combinations = 256)

Parameter	Q_{num}	N_{num}	λ	S_{max}	Z_{skew}	Ω_{max}	*Gen*
Values	250, 500	4, 8, 16, 32	10, 100	25, 50	0, 1	10, 100	*Nrm*, *Div*
# options	2	4	2	2	2	2	2

Table 3: WG_I vs WG_{TN} breakdown (for 256 experiments)

Max Cost	**Weave-Group Inserted (WG_I)**	**Weave-Group to Nodes (WG_{TN})**
Wins	Best in **80%** of cases	Best in **17%** of cases
Loses	Not best in 20% of cases, and within **3%** from the best on average	Not best in 83% of cases, and within **48%** from the best on average

Total Cost	**Weave-Group Inserted (WG_I)**	**Weave-Group to Nodes (WG_{TN})**
Wins	Best in **5%** of cases	Best in **90%** of cases
Loses	Not best in 95% of cases, and within **9%** from the best on average	Not best in 10% of cases, and within **0.2%** from the best on average

(a) **Max Cost** Comparison (b) **Total Cost** Comparison

Fig. 2: ***Average Plan Quality*** *(from 256 experiments) where* ***0%*** *and* ***100%*** *are the average plan costs of all* ***best*** *and* ***worst*** *plans, respectively, across all optimizers. The error bars show the standard deviations Consistent with the definition of a standard deviation, about 68% of all plans produced by these optimizers lie in this margin.*

similar quality. In some other cases where W_{TN} and W_I performed poorly, the optimizer *Group to Lowest* (G_{TL}) performed better. In such cases, our optimizers WG_{TN} and WG_I were still able to match the best plans produced by G_{TL} with equal or better quality plans in most of the cases. Thus, we concluded that the WG_{TN} and WG_I optimizers were able to successfully adapt to different environments and produce the best plans in terms of *Total Cost* and *Max Cost*, respectively.

To compare and contrast the two winning optimizers, we provide the breakdown of their performances in Table 3. From this table, we see that in terms of *Max Cost*, WG_{TN} significantly falls behind WG_I, since balancing is not the first priority of WG_{TN}. In

Table 4: Average Plan Generation Runtime (for 256 experiments)

Optimizer	G_{Rand}	W_{Rand}	WG_{Rand}	G_{RR}	W_{RR}	WG_{RR}	G_{TL}	W_{TL}	WG_{TL}	W_{TN}	WG_{TN}	W_I	WG_I
Time (sec)	0.01	2.31	0.02	0.01	2.34	0.01	0.01	12.6	9.11	2.95	2.83	5.68	3.94

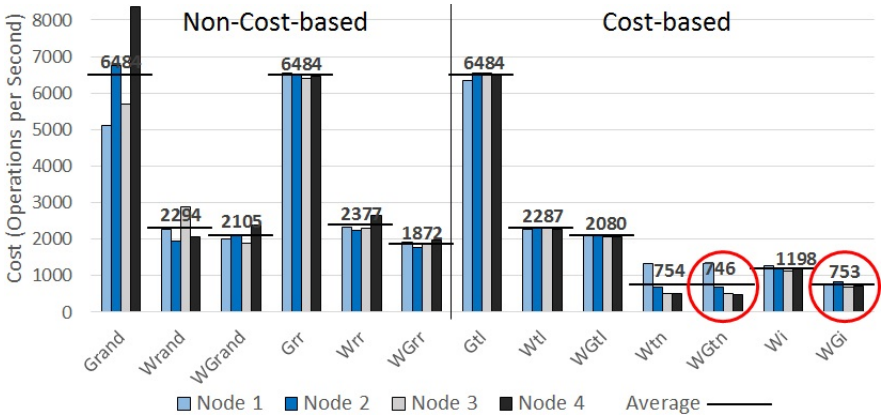

Fig. 3: Costs per node in a 4-node system

terms of *Total Cost*, WG_{TN} always either wins or is within 0.2%, and WG_I falls behind, but not as significantly, since it is on average within 9% of the winning optimizer.

Additionally, we have recorded the runtimes of our optimizers (Table 4), and we see that plan generation time on average does not exceed 13 sec per plan for all optimizers, which is fast considering that after an execution plan is generated and deployed to the *DSMS*, it is expected to run for a significantly longer time.

Take-away WG_{TN} and WG_I produce the best execution plans in terms of *Total Cost* and *Max Cost*, respectively. WG_{TN} falls behind WG_I in terms of *Max Cost* more significantly than WG_I falls behind WG_{TN} in terms of *Total Cost*. All optimizers generate plans fast (< 13 sec).

Experiment 2: Load Balancing

Configuration To show how all proposed algorithms compare in terms of balancing load and minimizing the total plan cost, we fix a few parameters ($Q_{num} = 250$, $N_{num} = 4$, $\lambda = 100$, $S_{max} = 25$, $Z_{skew} = 1$, $\Omega_{max} = 100$, $Gen = Nrm$) and run this experiment while recording the individual node costs of produced plans for all optimizers.

Results (Fig. 3). The results depict the typical behavior of the proposed algorithms in a 4-node environment. Since algorithms W_{TN} and WG_{TN} are optimized mostly for *Total Cost*, they produce plans with very imbalanced node loads. However, their *Total Costs* (as well as their Average Costs) are low. On the other hand, W_I and WG_I produce plans that are well balanced, and, at the same time, WG_I produces plans that also have a low *Total Cost* (practically as low as WG_{TN}).

Take-away Algorithms that are producing execution plans with the lowest *Total Cost* typically perform poorly in terms of balancing load among the different nodes.

8 Related Work

DSMSs have become the popular solutions to meet the near-real-time requirements of monitoring, as well as online analytics applications. As a result, the initial *DSMS* prototypes [4, 18, 5, 25, 22, 7] are replaced with research and commercial distributed *DSMSs* [24, 1, 5, 6, 2, 3]. In these systems, the techniques for the efficient processing of *ACQs* could be broadly classified into techniques for: 1) the *implementation* of the continuous aggregation operator, and 2) the *multi-query optimization* of multiple continuous aggregate queries. Although the focus of this paper is on the latter, we first briefly review the former for completeness.

Under the operator implementation techniques, *partial aggregation* has been proposed to minimize the repeated processing of overlapping data windows within a single aggregate (e.g., [16, 17, 10, 15, 27, 28]) by processing each input tuple only once. As discussed in Section 2, *ACQ* processing is typically 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. Recently, in order to minimize the cost of final aggregation, *TriOps* [11] uses an intermediate function between the sub-aggregation and final-aggregation levels to pipeline partial aggregate results to final-aggregate functions.

Under the multi-query optimization techniques, the general principle is to minimize (or eliminate) the repeated processing of overlapping operations across multiple aggregate queries. 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.

Techniques leveraging the overlaps in predicate conditions and group-by attributes across different *ACQs* are similar to classical multi-query optimization [21] that detects common subexpressions. Techniques leveraging shared processing of overlapping windows across different *ACQs* emerged with the paradigm shift for handling continuous queries. The *shared time slices* technique [15], for example, has been proposed to share the processing of multiple continuous aggregates with varying windows. It has also been extended into *shared data shards* in order to share the processing of varying predicates, in addition to varying windows. Orthogonally, [19] extends classical subsumption-based multi-query optimization techniques towards sharing the processing of multiple *ACQs* with varying group-by attributes and similar windows.

Like *shared time slices*, *WeaveShare* [12] addresses the problem of shared processing of aggregate queries with varying windows. *WeaveShare*, however, employs a novel *Weavability* metric that allows selective partitioning of the *ACQ* workload into multiple, disjoint execution trees resulting in a dramatic reduction in processing costs.

Weavability is also the underlying principle of our work in this paper, which we utilize to achieve scalability in distributed environments. Unlike our work, which is based on multiple query optimization, other work that addresses distributed processing of *ACQs* is based on MapReduce [9]. In [9], a demonstration of implementing event monitoring applications using the modified Hadoop framework was presented. Along the same lines are schemes for scaling operators/queries out when nodes get overloaded [13, 14], but these do not focus on load balancing and combining *ACQs* as in this paper.

With respect to load balancing, the underlying principle of the cost-based *ACQ* assignment to nodes in our work is similar to the basic greedy approach in Operating System where a process is assigned to execute on the currently least loaded node [20].

9 Conclusions

In this paper, we explored how the sharing of partial aggregations can be done in the environment of distributed *DSMSs*. We formulated the problem as a distributed multi-*ACQs* optimization which combines sharing of partial aggregations and assignment to servers to produce high quality plans that keep the total cost of the execution low and balance the load among the computing nodes. We presented a classification of optimizers based on whether they are cost-based and how they utilize the concept of *Weavability*. We implemented and experimentally compared all of our proposed optimizers.

Our evaluation showed that the *Weave-Group Inserted* (WG_I) optimizer delivers the best quality in terms of load balancing, which makes it the most beneficial for *Cloud* service providers, since balancing helps conserve energy and prevents the need to over-provision systems hardware. At the same time, our evaluation showed that the *Weave-Group to Nodes* (WG_{TN}) optimizer best minimizes the total plan cost, which makes WG_{TN} the most beneficial for clients, since the monetary cost of *ACQ* computation in multi-tenant environments becomes lower. A closer look at the performance profiles of the two winning optimizers suggests that it might be more advantageous to choose the WG_I optimizer in the case where both service providers and clients should be satisfied "equally" – WG_I falls behind in terms of *Total Cost* less significantly (only 9% on average) than WG_{TN} does in terms of *Max Cost* (load balancing).

Currently, we are looking at extending our work in (1) heterogeneous environments, where nodes have different computational capacities, (2) dynamic environments, where *ACQs* and nodes can be added/removed on-the-fly, and (3) evolving workloads, where the input rate fluctuates as well as the background system utilization. In the future, we are planning to address *ACQ* optimization as part of general multi-query optimization of CQs with overlapping predicate conditions and group-by attributes.

Acknowledgments We would like to thank Cory Thoma, Nikolaos Katsipoulakis, and the anonymous reviewers for the insightful feedback and Mark Silvis for his help with copyediting. This work was supported in part by NSF award CBET-1250171, a gift from EMC/Greenplum and an ACM SoCC 2015 Student Scholarship.

References

1. Apache samza. http://samza.apache.org.
2. S4 distributed stream computing platform. http://incubator.apache.org/s4.
3. Spark streaming. https://spark.apache.org/streaming.
4. 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. *VLDBJ*, 12:120–139, 2003.
5. T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *VLDB*, 6:1033–1044, 2013.

6. R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.
7. P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, pages 1–3, 2010.
8. C. Chung, S. Guirguis, and A. Kurdia. Competitive cost-savings in data stream management systems. In *COCOON*, pages 129–140. 2014.
9. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, pages 1115–1118, 2010.
10. 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. *TKDE*, 19:57–72, 2007.
11. S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.
12. S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, pages 357–368, 2011.
13. V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 23:2351–2365, 2012.
14. N. R. Katsipoulakis, C. Thoma, E. A. Gratta, A. Labrinidis, A. J. Lee, and P. K. Chrysanthis. Ce-storm: Confidential elastic processing of data streams. In *SIGMOD*, pages 859–864, 2015.
15. S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.
16. 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 Rec.*, 34:39–44, 2005.
17. J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.
18. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
19. K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, pages 852–863, 2011.
20. H. E. Romeijn and D. R. Morales. A class of greedy algorithms for the generalized assignment problem. *Discrete Applied Mathematics*, 103:209–235, 2000.
21. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.
22. M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *TODS*, 33:1–44, 2008.
23. A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. F1: Accelerating the optimization of aggregate continuous queries. In *CIKM*, pages 1151–1160, 2015.
24. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *SIGMOD*, pages 147–156, 2014.
25. Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.
26. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS*, pages 374–382, 1995.
27. R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.
28. R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *VLDBJ*, 19:557–583, 2010.