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 on-line 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 produced. A large volume of this generated data often takes the form of high velocity streams. At the same time, on-line 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, 23, 5, 24, 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 on-line analytics can be found in stock market web applications where multiple clients monitor price fluctuations of stocks. In these settings,

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 levels in terms of accuracy.

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 re-calculated. 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 period 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.

ACQs require the *DSMS* to maintain their state over time, while performing aggregations. It is clear that *ACQs* with a greater accuracy 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 *Cloud* infrastructures to minimize the purchase and maintenance cost of machinery and to be able to scale their services. This deployment of *DSMSs* to *Cloud* infrastructures results in multi-tenant settings, where multiple *ACQs* with diverse periodic properties are executed on the same hardware.

Problem Statement It is safe to say that today most *Cloud* systems are utilizing distributed processing environments with multiple multi-core computing nodes as their primary service infrastructures. The efficiency of such environments 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 on-line 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 [26], 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 very 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]. Being targeted for single-node *DSMSs*, *WeaveShare* is oblivious to distributed processing capabilities, and as our experiments have revealed, *WeaveShare* cannot produce high quality execution plans that assign the combined *ACQs* to the various comput-

ing 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 plans per node, such that it keeps the total cost of the *ACQs* execution low and balances the load among the computing nodes.

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

- *Minimizing the total cost* of the execution plan allows the system to support more client requests. 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 adopts an existing strategy or develops 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, and to 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 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.

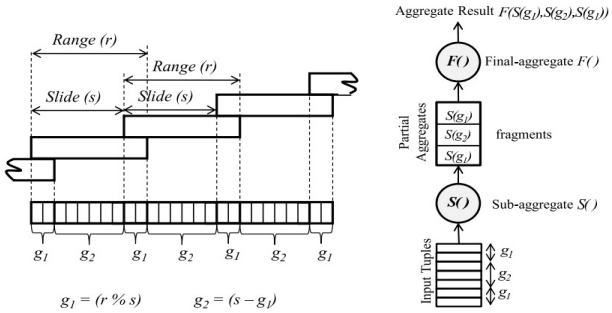


Fig. 1: Paired Window Technique

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 There are two *ACQs* that perform the *count* aggregate operation 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. Therefore, 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 the location is already marked, it cannot be marked again. 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 either use *Bit Set* [12] or *Formula F1* [22] techniques.

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 was a good choice or not.

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

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 individual execution tree. Then, it iteratively considers all possible pairs of execution trees and combines those which most reduce the total plan cost into a single tree. *WeaveShare* produces a final execution plan 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 located anywhere on the Internet 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* which 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_{TN}	W_I
	Weave + Group	WG_{RAND}	WG_{RR}	-	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 lowers 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:

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

This metric is important for measuring the load balance in execution plans generated by different *ACQs* optimizers. Load balancing is essential for the *Cloud* environment because keeping the load balanced between operational nodes directly translates into energy savings and thus monetary savings. Significant energy is saved if the computing nodes run at the lowest possible frequency given that the operational load directly correlates to the processor frequency, and energy consumption is at least a quadratic function of processor frequency [25].

Additionally, 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.

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 first class of optimizers: *Non-Cost-based* optimizers.

5.1 Group Only

- Group Random (G_{RAND}): the optimizer first randomly selects a node for each inserted *ACQ* and adds it to the selected node as a separate execution tree.
- Group Round Robin (G_{RR}): the optimizer selects a node for each inserted *ACQ* one by one (in a Round Robin fashion) and adds it to the selected node as a separate execution tree.

5.2 Weave Only

- Weave Random (W_{RAND}): the optimizer first randomly selects a node for each inserted *ACQ* and *weaves* it into the single execution tree on the selected node.
- Weave Round Robin (W_{RR}): the optimizer selects a node for each inserted *ACQ* one by one (in a Round Robin fashion) and *weaves* it into the single execution tree on the selected node.

5.3 Weave and Group

- Weave-Group Random (W_{RAND}): the optimizer first randomly selects a node for each inserted *ACQ* and randomly chooses whether to add it as a separate tree, or to *weave* it with one of the available trees on the selected node.

- Weave-Group Round Robin (W_{RR}): the optimizer selects a node for each inserted *ACQ* one by one (in a Round Robin fashion) and *weaves* it into the next execution tree in line on a selected node, or adds it as a separate tree when the number of *ACQs* in each tree becomes equal to the current number of trees on this node.

6 Cost-based Optimizers

In this section, we provide the details on the second class of optimizers: *Cost-based* optimizers.

6.1 Group Only

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.
Algorithm: The G_{TL} optimizer adds each *ACQ* to a separate execution tree, and then all of these trees are grouped into the available nodes in a cost-balanced fashion. The trees are first sorted by their costs, and then, starting from the most expensive trees, they are assigned to the nodes that currently have the lowest total cost (the initial node cost is zero).
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).

6.2 Weave Only

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 when the current number of trees is less than or equal to the number of available nodes, W_{TN} stops and finalizes the current plan. At this point, we already have the correct number of trees that can be equally distributed between the available nodes. 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 less than or equal to the number of available nodes. Thus, W_{TN} forces *WeaveShare* to generate the specific number of execution trees equal to the number of nodes.
Discussion: Since W_{TN} is a direct descendant of *WeaveShare*, it is also optimized to produce the minimum *Total Cost*. The downside of this plan is that it does not perform any load balancing, which causes generated plans to not be suitable for the distributed systems where the capacity of the most powerful CPU is lower than the cost of the most expensive execution node (in terms of computational load).

Additionally, since W_{TN} allows only one execution tree per node, it forces *WeaveShare* to keep merging trees with very different sets of *ACQs*, which results in combining *ACQs* that do not *weave* well together. This consequently causes larger plan costs.

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*.
Algorithm: The difference of 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, and W_I is optimized for the *Max Cost* instead of the *Total Cost*.

W_I works by iterating through all *ACQs* that need to be installed and assigning them temporarily to each available node (which has a single tree). The cost of each node is kept, and for each node we calculate the new node cost with the next *ACQ* inserted. Each *ACQ* is then permanently assigned to the node that has the smallest new cost. This way, W_I takes into account both the *Weavability* of the inserted *ACQ* with every available node, and performs cost-balancing of the computation nodes.
Discussion: The downside of this optimizer is that, since load balancing is the first priority of W_I, it sometimes assigns *ACQs* to nodes with underlying trees that they do not *weave* well with. 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 causes generated plans to have higher *Total Costs*.

6.3 Weave and Group

Weave-Group to Nodes (WG_{TN}) Like $W_{TN}q$, 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 stops execution if it reaches the point when 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 is still greater than the number of available nodes, the WG_{TN} plan takes all current execution trees, and groups them between the available nodes without *weaving* them. The grouping happens in a balanced fashion. First, all trees are sorted by their costs, and starting from the most expensive ones, the trees are assigned to the computation nodes with the smallest current total cost (the initial cost of each node is zero). When all of the trees are grouped to nodes, WG_{TN} returns the final execution plan.
Discussion: Similarly to W_{TN}, the WG_{TN} optimizer is also designed to produce the minimum *Total Cost*. However, it also attempts to achieve balancing by sorting and grouping the produced execution trees. Unfortunately, since these execution trees are sometimes of significantly different costs, this load balancing technique does not always produce the desired output.

Also, since unlike W_{TN}, WG_{TN} does not force trees that do not *weave* well together to merge, it can achieve a better *Total Cost* in many cases. Since WG_{TN} groups these trees to nodes without *weaving* them together, the total cost of the plan does not increase. However, the penalty of grouping trees on nodes without merging them is that the input rate is now multiplied based on the number of the trees on each node. This

happens because input tuples will have to be processed as many times as there are separate execution trees on each node. Clearly, the higher the input rate of the stream, the more costly it will be for the system to group trees without *weaving*.

Weave-Group Inserted (WG_I) This optimizer is also a version of the *Insert-then-Weave* approach optimized for the *Max Cost* (but in the *Weave and Group* category). Since the WG_I optimizer does not have to be limited by only one execution tree per node, it utilizes grouping to keep the *Total Cost* low while maintaining load balance between nodes.
Algorithm: WG_I keeps the current cost of each node similarly to W_I. The difference is that since we can have multiple trees per node, all trees from all nodes have to be tested with each incoming *ACQ* in order to find the node with the smallest cost after *weaving* this *ACQ* into the best tree of this node. After finding the node with the smallest new cost, WG_I compares it to the cost of the incoming *ACQ* running in a separate tree. If the new cost of inserting this *ACQ* into the existing tree is smaller, the *ACQ* becomes permanently merged into this tree. Otherwise, a new tree for this *ACQ* is created and added (grouped) to the node which has the smallest current cost. The optimization continues until all *ACQs* are assigned.
Discussion: Even though the WG_I optimizer uses grouping execution trees on different nodes, 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 current incoming *ACQ* currently has the smallest cost and is located in the node with the smallest current node cost.

Note The preprocessing step can be done for the **four** optimizers mentioned above: W_{TN}, W_I, WG_{TN}, and WG_I. It is done by merging all *ACQs* with identical slides into trees, since such *ACQs weave* together the best. Then, our workload consists of not only single *ACQs*, but also of trees with *ACQs* with the same slides. The optimizers iterate over this workload the same way as before. Note that this preprocessing is only beneficial in terms of the *Max Cost* for the distributed systems with low numbers of nodes compared to the numbers of *ACQs*. Otherwise, since the workload now has fewer entities, it is harder to balance them among the higher number of computation 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}). We assume that all *ACQs* are installed on the same data stream and we can share partial aggregations between them.
- Number of nodes in the target system (N_{num}) for which we are producing execution plans.
- The input rate (λ), which describes how fast tuples arrive through the input stream in our system.
- 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 our workload is normal (*Nrm*) and includes any slides or diverse (*Div*), and 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.

All results are taken as averages of running each test three times. We ran all of our experiments on a 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 11 of the above mentioned optimizers on this workload. For each experiment we took the average of three runs.

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

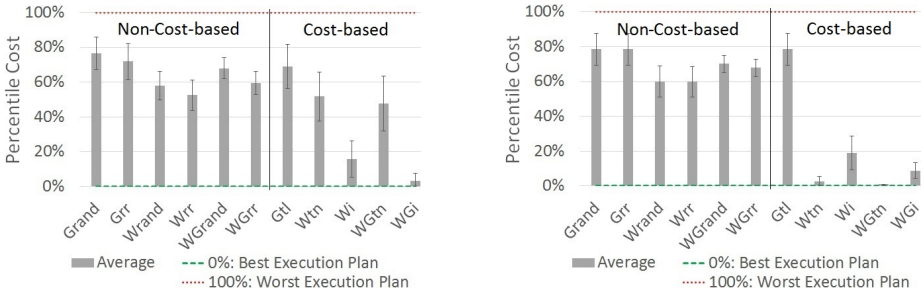
Table 2: Experimental Parameter Values (Total number of combinations = 256)

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* ***winning*** *and* ***loosing*** *plans, respectively, across all optimizers. The error bars show the standard deviations of the optimizers' performances. Consistent with the definition of a standard deviation, about 68% of all plans produced by these optimizers lie in this margin.*

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 when 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 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

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

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

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 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 6 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 (< 6 sec).

Experiment 2: Load Balancing

Configuration To show how all proposed algorithms compare in terms of balancing load and in terms of 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 execution plans for all the above mentioned 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

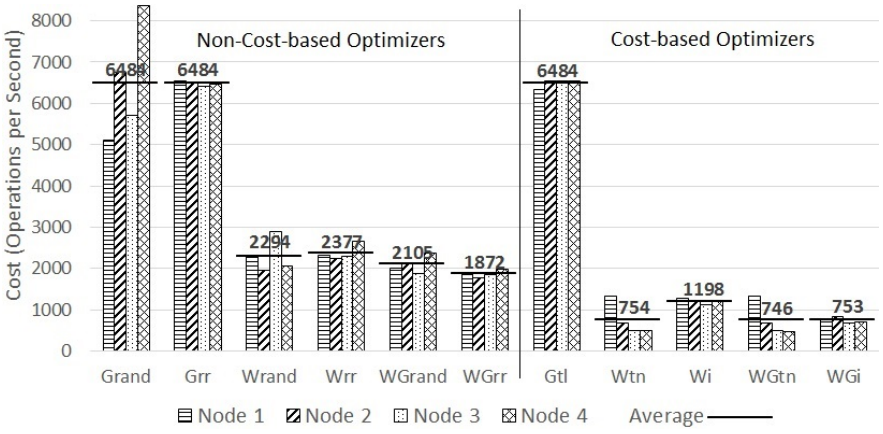


Fig. 3: Costs per node in a 4-node system

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 on-line analytics applications. As a result, the initial *DSMS* prototypes [4, 18, 5, 24, 21, 7] are replaced with research and commercial distributed *DSMSs* [23, 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.

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, 26, 27]) 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 [20] 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 continuous aggregate queries 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 the optimizer to selectively partition the aggregate continuous queries workload into multiple, disjoint execution trees resulting in a dramatic reduction in total 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 combining queries as presented in this paper.

9 Conclusions

In this paper, we explored how the sharing of partial aggregations can be implemented in the more challenging processing 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 *ACQs* execution low and balance the load among the computing nodes. We presented a classification of optimizers based on whether or not they are cost-based and on 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* optimizer delivers the best quality in terms of load balancing among the nodes in the system, 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* optimizer best minimizes the total plan cost, which makes *Weave-Group to Nodes* 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 *Weave-Group Inserted* optimizer in the case where both service providers and clients should be satisfied "equally" – *Weave-Group Inserted* falls behind in terms of *Total Cost* less significantly (only 9% on average) than *Weave-Group to Nodes* does in terms of *Max Cost* (load balancing).

Acknowledgments We would like to thank C. Thoma and the anonymous reviewers for the insightful feedback. This work was supported in part by NSF award CBET-1250171 and a gift from EMC/Greenplum.

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*, 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. In *VLDB*, 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*, 2013.
7. P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, 2010.
8. C. Chung, S. Guirguis, and A. Kurdia. Competitive cost-savings in data stream management systems. In *COCOON*. 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*, 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*, 2007.
11. S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, 2012.
12. S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.
13. V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 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*, 2015.
15. S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 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.*, 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*, 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*, 2011.
20. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
21. M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *TODS*, 2008.
22. A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Accelerating the optimization of aggregate continuous queries. In *CIKM*, 2015.
23. 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*, 2014.
24. Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, 2005.
25. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS*, 1995.
26. R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.
27. R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *VLDBJ*, 2010.