Competitive Cost-Savings in Data Stream Management Systems

Christine Chung *, Shenoda Guirguis **, and Anastasia Kurdia ***

Abstract. In Continuous Data Analytics and in monitoring applications, hundreds of similar Aggregate Continuous Queries (ACQs) are registered at the Data Stream Management System (DSMS) to continuously monitor the infinite input stream of data tuples. Optimizing the processing of these ACQs is crucial in order for the DSMS to operate at the adequate required scalability. One optimization technique is to share the results of partial aggregation operations between different ACQs on the same data stream. However, finding the query execution plan that attains maximum reduction in total plan cost is computationally expensive. *Weave Share*, a multiple ACQs optimizer that computes query plans in a greedy fashion, was recently shown in experiments to achieve more than an order of magnitude improvement over the best existing alternatives. Maximizing the benefit of sharing, i.e., maximizing the cost-savings achieved by sharing partial aggregation results, is the goal of Weave Share. In this paper we prove that Weave Share approximates the optimal cost-savings to within a factor of 4 for a practical variant of the problem. To the best of our knowledge, this is the first theoretical guarantee provided for this problem. We also provide exact solutions for two natural special cases.

1 Introduction

In Continuous Data Analytics, such as pay-per-click applications, and in monitoring applications, such as network, financial, health and military monitoring, hundreds of similar Aggregate Continuous Queries (ACQs) are typically registered to continuously monitor unbounded input streams of data updates [13, 7]. For example, a stock market monitoring application allows each of its numerous users to register several monitoring queries. Traders interested in a certain stock might register ACQs to monitor the `average`, or `maximum`, trade volume in a certain period of time, e.g., the last 1, 8, or 24 hours. Meanwhile, decision makers might register monitoring queries for analysis purposes with coarse time granularity over the same data stream, e.g., the `average` trade volume in last week or month. Given the high data arrival rates, optimizing the processing of ACQs is crucial for scalability of the system. Data Stream Management Systems (DSMSs) were developed to be at the heart of every monitoring application, (e.g., [3, 9, 2, 1, 14, 15]). DSMSs must efficiently handle the unbounded streams with large volumes of data and large numbers of continuous queries. Thus, devising ways to optimize the processing of multiple continuous queries is imperative for DSMSs to

* Department of Computer Science, Connecticut College. *cchung@conncoll.edu*
** Oracle Inc. *shenoda.guirguis@oracle.com*
*** Department of Computer Science, Bucknell University. *ak034@bucknell.edu*

exhibit the scalability required. The commonality of many of the ACQs is what makes optimization possible.

An ACQ is typically defined over a certain window of the input data stream, to bound its computations. (For example, an ACQ that monitors the average trade volume of a stock index could report every hour the average trade volume in the past 24 hours). Partial aggregation has been proposed to optimize the processing of an ACQ [11, 12, 5] by minimizing the repeated processing of overlapping windows. Partial aggregation has also been utilized to share the processing of multiple similar ACQs with different windows [10, 13, 7, 8]. Recently, the concept of *Weaveability* of two sets of ACQs was introduced as an indicator of the potential benefits of sharing their processing [7]. By exploiting weavability, the algorithm *Weave Share* optimizes the shared processing of ACQs. *Weave Share* considers all factors that affect the cost of the shared query plan. It selectively groups ACQs into multiple "execution trees" to minimize the total plan cost. It was shown experimentally that *Weave Share* generates up to 40 times better quality plans compared to the best alternative sharing scheme [7].

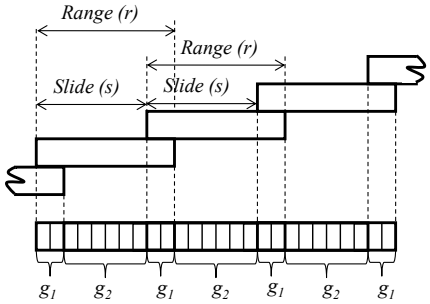
Contributions In this paper we provide formal guarantees on the performance of the Weave Share algorithm. The total cost of a query plan can be represented as the cost of the no-share query plan, in which all partial aggregations are independent, minus the cost-savings achieved by sharing some partial aggregation operations. We provide a lower bound for the cost-savings achieved by Weave Share. Specifically, we show that for a widely applicable variant of the problem, in the worst case, Weave Share is guaranteed to achieve a cost-savings of at least $\frac{1}{4}$ of the maximum possible cost-savings. In contrast with total cost of the query plan, cost-savings is actually a more incisive measure that removes the distraction of the minimum "base-cost" that exists for any given instance, even under the most optimal sharing arrangement. In the Appendix, we also provide exact solutions for two practical special cases of the problem.

2 Background and definitions

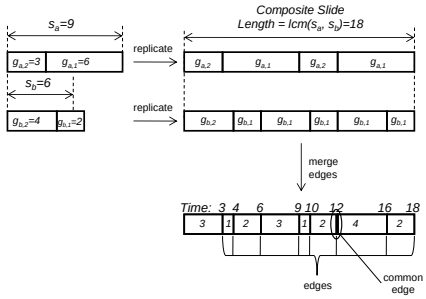
We set the stage by providing the necessary background on ACQ semantics, the paired window technique for partial aggregation [10], the procedure for "composing" multiple ACQs together [10] so that the results of their partial aggregation can be "shared." We then give a formal definition of the optimization problem at hand and give an overview of an efficient practical algorithm for the problem.

2.1 Partial aggregation for one ACQ

Each ACQ (or *query*) comprises an aggregation operator (`sum`, `max`, `count`, etc.) along with two parameters: the *range* r, the length (in time) of the window of data being aggregated, and the *slide* s, which indicates how frequently the results should be reported. For example, an ACQ may request that the maximum price of a particular stock over the last hour ($r = 60$ min) be reported every 10 minutes ($s = 10$ min). If $r > s$, as is the case in this example, we have a *sliding* or *overlapping* window, where



(a) The paired window technique.



(b) Creation of a composite slide for 2 ACQs, a and b, with respective ranges of 12 and 10 seconds.

a single data tuple belongs to more than one window. If a new stock price value is generated every minute, then this tuple participates in six aggregation operations.

Rather than aggregate the entire window of tuples from scratch each time, *partial aggregation* [11, 10] first computes sub-aggregations of successive pieces of the window, then applies a *final aggregation* function over these sub-aggregates. For example an aggregate `count` would be computed by first using a `count` on each part of the window, then using a `sum` over the partial counts. This technique can be used over all the distributive functions widely used in database systems. It reduces query processing cost by preventing tuples from having to be aggregated repeatedly. Instead, each input tuple is processed once by the *sub-aggregation operator*, the result of the sub-aggregation gets buffered, and the final aggregate is assembled from those partial aggregates.

To capitalize on the idea of partial aggregation, Krishnamurthy et al. [10] proposed the *paired window* technique, whereby each *slide* is partitioned into at most two slices or *fragments* g_1 and g_2. See Figure 1a for an illustration. As pictured, $g_1 = r \mod s$, and $g_2 = s - g_1$. Thus, since r/s is the number of slides per window, computing each final aggregation uses at most $\lceil 2r/s \rceil$ operations. This paired window approach allows for effective sharing of the partial aggregation results for different queries on the same data, the details of which we describe in the following section.

2.2 Merging multiple ACQs

To process multiple ACQs with different range and slide parameters, there are two basic strategies [10]: *unshared partial aggregation* (also referred to as *no-sharing*), or *shared partial aggregation*. When unshared partial aggregation is used, each query is simply processed separately using the paired window technique described above. This requires storing multiple copies of the input tuples, as each query is answered using its own individual sub-aggregation results.

For the shared partial aggregation strategy for $k > 1$ ACQs, we need to compute the fragments on which partial aggregation is applied in a different manner, so that the sub-aggregation results can be reused in computing a different final aggregation for each query. These fragments are computed as follows. For k ACQs with slides

$s_1, s_2, \ldots, s_k$, we create a *composite slide* of length s' equal to the least common multiple of $(s_1, s_2, \ldots, s_k)$. Copy and repeat each slide $i = 1 \ldots k$ along with its corresponding paired fragments s'/s_i times, to fit the full length of the composite slide s' (see Figure 1b). The end of each fragment is referred to as an *edge*, and it serves as a demarcation of the boundary between two fragments. (We use the term *edge* in this paper to ensure consistency with previous work on query planning).

The edges or fragments of the final composite slide are determined by the distinct edges that remain after all k slides have been overlayed. After these new fragments are computed, partial aggregation can be applied on each fragment, and the results can be shared between different ACQs. While shared partial aggregation certainly reduces processing costs at the sub-aggregation level, it increases costs at final aggregation level. Depending on the queries, the overall total cost may be higher than in case of no sharing at all (see Example 1 in the Appendix).

2.3 The objective function

An input instance for our optimization problem is comprised of a set of n ACQs. We will say that two or more ACQs are *shared* if their partial aggregation was shared via the shared partial aggregation strategy described above. We define an *execution tree* to be a subset of the n ACQs in which all the ACQs in the subset are shared. A *query plan* Q is then a grouping of the n ACQs into m execution trees, $t_1, t_2, \ldots, t_m$.

For each tree t_i, let E_{t_i} (or, more simply, E_i) be the number of fragments generated per second (also referred to as the *edge rate*). Let Ω_i denote the total number of final-aggregation operations performed on each fragment, which is termed the *tree overlap factor*. If tree i consists of k shared queries, we can compute

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

The *cost* of processing a single tree t_i, in terms of the total number of aggregate operations per second, is thus

$$C(t_i) = \lambda + E_i \Omega_i \tag{1}$$

where λ is the number of tuples arriving per second (tuple input rate) that represents the cost at sub-aggregation level.

And the *total cost* of a query plan Q with m trees is simply the sum of the costs of the individual trees.

$$C(Q) = m\lambda + \sum_{i=1}^{m} E_i \Omega_i \tag{2}$$

$C(Q)$ represents the total number of aggregations per second for all ACQs. The formal problem statement is then as follows.

> *Given a set of n ACQs, find a query plan (a partitioning of the ACQs) that minimizes total cost.*

To recap, the cost of one execution tree comprises two parts: the cost of the sub-aggregations (at the intermediate aggregation step) *plus* the cost of final aggregations at the final aggregation step. When queries in two trees are merged into one execution tree, the number of necessary intermediate (sub) aggregations decreases, because partial aggregates that previously needed to be computed for both trees independently, now can be computed just once and reused in answering both queries. However, the number of final aggregation operations increases: first, because the number of edges per second (edge rate) in the resulting execution tree will be at least the maximum edge rate of individual trees, and second, because now the final aggregations will need to be performed for each edge, for each query. The goal of sharing processing of queries is to maximize savings during the sub-aggregation step while minimizing the costs at the final-aggregation step.

2.4 Weave Share

Weave Share (WS) is a recently proposed greedy heuristic algorithm for computing query plans [7]. Guirguis et al. first formalized the notion of *weavability* of multiple queries as the ratio of the number of edges common to multiple ACQs to the total number of edges in the composite slide. When several queries share partial aggregation operations, the more common edges between ACQs that exist in their composite slide, the more weavable they are. Naturally, to maximize the benefit of sharing ACQs, either the shared ACQs should exhibit a high degree of weavability or the total overlap factor (number of total final aggregation operations performed on shared fragments) should be low, or both. Weave Share considers both of these factors in optimizing the processing of ACQs. Each iteration of WS involves one *merge* step, where the queries of two separate execution trees are combined into one tree, so that the two previously separate groups of shared queries are now all shared together in one execution tree. There may be up to $n - 1$ iterations, and the total time required by the WS algorithm is $O(n^2)$, where n is the number of ACQs.

The Weave Share Algorithm The input to the algorithm is the original set of n ACQs, and the output is a query plan, or a partition of the ACQs into $m \leq n$ disjoint groups (execution trees).

1. Create n trees, one ACQ per tree.
2. Consider all possible pairs of trees. For each pair of trees, compute the reduction in cost that would be achieved if queries belonging to both trees were merged.
3. Find the maximum reduction in cost over all possible pairs of trees. Ties may be broken arbitrarily.
4. If this value is positive (i.e. it is indeed a cost-reduction), merge these trees and repeat from step 2. Otherwise (the value is not positive), terminate the algorithm.

Experimental Performance of Weave Share Experimental results by Guirguis et al. [7] demonstrate that the query plans produced by Weave Share outperform query plans generated by other common algorithms, such as *No Share* (in which partial aggregation results are not shared), *Random* (in which random trees are iteratively merged until

there is no longer an improvement), *Local Search* (that explores the solution subspace by starting with a random partition of ACQs into trees and iteratively moving single ACQs between trees), *Shared* (in which all queries constitute a single execution tree) and *Insert-then-Weave* (in which each individual query is inserted one-by-one into the tree that it weaves best with). For different input parameters (λ and n) the Weave Share plan had a cost as much as 40 times better than other plans. For small problem instances, exhaustive search was used to find the optimal query plan; Weave Share was able to find these plans in all but one instance.

The evaluation of Weave Share's performance was conducted on a synthetic data stream, to allow control over input parameters and cover the most likely real scenarios. Although Weave Share is shown to outperform other strategies on a synthetic data stream, there is no guarantee on the quality of the solution produced by the algorithm. Extensive comparison of Weave Share with an optimal solution produced by exhaustive search is not feasible for any practical number of ACQs. In short, Weave Share performs better than the alternative heuristic algorithms but not much is understood about how many more aggregations per second the query plan produced by Weave Share requires compared to the number of aggregations per second in an optimal query plan.

3 A Cost-savings Approximation

In this section, we give a guarantee on the amount of cost-savings that Weave Share achieves. The outcome of the Weave Share algorithm features a decrease in the total cost of the query plan compared to the *no-share* query plan in which each query is executed by itself. It is this improvement, or *savings*, achieved by the Weave Share algorithm that we seek to bound. We find this measure of maximizing cost-savings appealing, as it focuses on the achievements of the algorithm compared with that of the optimal solution. In contrast, under the umbrella of the objective of minimizing total cost, we would include costs that are inherent and unavoidable, to both WS and OPT.

We first introduce some notation. Recall that a query plan is a partitioning of the queries into execution trees. We refer to an execution tree that consists of more than one query as a *multi-tree*. We will indicate an execution tree by listing its ACQs in square brackets, for example: $[q_1, q_2, \ldots, q_k]$ refers to the multi-tree composed of queries $q_1, \ldots, q_k$ merged together, and $[q_i]$ refers to an execution tree with a single stand-alone query q_i. We can indicate a query plan using a set of such lists. For example, $\{[q_1], [q_2, q_3], [q_4]\}$ is a query plan with three execution trees, one with only query q_1, another with two merged queries q_2 and q_3, and another with only query q_4.

Let Q denote the query plan produced by the Weave Share algorithm, and m denote the number of execution trees in Q. Let Q^* denote an optimal query plan (one that minimizes total cost over all query plans), and m^* denote the number of its execution trees. Let N denote the no-share query plan, which has n execution trees, one for each stand-alone ACQ. Let μ be the number of merges made by WS to reach Q, and μ^* be the number of merges required to reach Q^* from N. Note that

$$n = m + \mu = m^* + \mu^*, \tag{3}$$

since each separate tree represents a potential merge that did not take place.

For any query plan X we denote the difference between the cost of the no-share plan and the cost of X by $R(X) = C(N) - C(X)$. Note that if $R(X)$ is positive, it represents a reduction, or savings, in cost compared to the no-share plan.

We are now ready to state the central theorem of this work.

Theorem 1. *The amount of savings achieved by the Weave Share query plan is at least a quarter of the savings achieved by an optimal query plan, i.e.,*

$$R(Q) \geq R(Q^*)/4,$$

under the following three conditions:

- *the range and slide of each query coincide (i.e., $r_i = s_i$ for each query q_i)*
- *the tuple input rate λ exceeds twice the edge rate of any tree comprised of exactly two ACQs (i.e., $\lambda \geq 2E_{[q_i,q_j]}$ for any $i, j \in \{1, \ldots, n\}, i \neq j$)*
- *the number of merges made by WS in Q is at least half the number of merges made in Q^* (i.e., $\mu \geq \mu^*/2$)*

We note that the special case of the problem enforced by the above three conditions is quite natural and applicable to practical settings. It is common in real-world applications for query windows to "tumble" [10], with disjoint windows that cover the entire input. The assumption on the size of λ is quite modest for most practical settings. And the third condition, after applying equation (3), is equivalent to $m \leq n/2 + m^*/2$. This condition has easily held in all previous WS experiments [7, 6] that have been conducted.

To prove this theorem, we first establish two useful lemmas, both of which provide interesting insight into the nature of sharing and merging.

3.1 Savings dilution

Our first lemma demonstrates that under some practical assumptions, there is some degree of "dilution" in cost-savings when merging a query into a multi-tree compared with merging two stand-alone queries. Specifically, we consider the restricted version of the problem where for each query its range and slide coincide ($r = s$). Moreover, assume that the tuple input rate λ exceeds twice the edge rate of any tree comprised of exactly two individual queries, i.e. $\lambda \geq 2E_{[a,b]}$ for any two queries a and b. Under these circumstances, we show that the savings achieved by merging two individual queries together is at least half the savings achieved by merging an individual query into a multi-tree.

Let $C([a], [b])$ denote the total cost of processing the two queries a and b separately, with no sharing. Let $C([a, b])$ denote the total cost when processing the two queries a and b as one, merged, execution tree (sharing their partial aggregates). Let $r([a], [b]) = C([a], [b]) - C([a, b])$, the cost-savings, or reduction in cost, from merging the two trees $[a]$ and $[b]$. More generally, let $r(t_1, t_2)$ denote the cost-savings from merging the two trees t_1 and t_2, either or both of which may be a multi-tree.

Lemma 1. *Consider an execution tree of k queries $[q_1, q_2, \ldots, q_k]$. Assume $r_i = s_i$ for all queries $i = 1 \ldots k$. Further assume $\lambda \geq 2E_{[q_i,q_j]}$ for any $i, j \in \{1, \ldots, k\}, i \neq j$. Let S be any subset of $\{1, \ldots, k\}$. For any $a, b, c \in S$,*

$$2r([q_a], [q_b]) \geq r([q_{S-c}], [q_c]),$$

where q_{S-c} denotes the multi-tree comprised of all queries in S other than c.

Proof. To improve readability, we will write E_a for $E_{[q_a]}$ and E_{ab} for $E_{[q_a,q_b]}$. We will similarly simplify the Ω terms. Since $r_i = s_i$ for all i, we have $\Omega_a = \Omega_b = \Omega_c = 1$. We begin by noting that the savings achieved by merging q_a and q_b is $r([q_a], [q_b]) = \lambda + E_a + E_b - 2E_{ab}$. The savings achieved by merging q_c and q_{S-c} is $r([q_{S-c}], [q_c]) = \lambda + (|S| - 1)E_{S-c} + E_c - |S| \cdot E_S$.

Note that using our assumption about λ, we have $\lambda + E_S \geq E_c + 2E_{ab}$. Hence, since $E_a + E_b \geq E_{ab}$, we can say $\lambda + 2(E_a + E_b) - 4E_{ab} \geq E_c - E_S$. From this we can conclude, using $E_S \geq E_{S-c}$, that $2(\lambda + E_a + E_b - 2E_{ab}) \geq \lambda + (|S| - 1)E_{S-c} + E_c - |S|E_S$, or equivalently, that $2r([q_a], [q_b]) \geq r([q_{S-c}], [q_c])$.

3.2 The OPT-sequence

Our second lemma will involve a careful WS-based specification of Q^*, our optimal query plan, which will ultimately allow us to map the merges made by WS to increments of savings in cost achieved by Q^*. We begin with a helpful observation.

Observation 2 *Consider an execution tree consisting of several ACQs. The total cost of this tree, and thus the savings achieved by merging the constituent queries together into the tree, does not depend on the order in which queries are merged into the tree.*

For an illustration, refer to the Example 2 in the Appendix.

Construction of OPT-lists Based on the above observation, any valid query plan, including Q^*, can be represented as a specific sequence of steps whereby each step constitutes merging an individual query either with another individual query or into a tree of queries. In other words, each tree of Q^* can be represented as an ordered list of constituent queries, and the order of list elements (from right to left) defines a specific order of merges for that tree. For instance, for four queries a, b, c, and d, we can use the ordered list $[q_a, q_b, q_c, q_d]$ to denote the execution tree that is comprised of all four queries, as well as to indicate the ordered steps of first merging q_c with q_d, then merging q_b with the tree $[q_c, q_d]$, then merging q_a with $[q_b, q_c, q_d]$.

Below, we describe a procedure for constructing an ordered query list for each tree of Q^*. We will refer to each ordered list of queries as an OPT-*list*, and the execution tree to which it refers as an OPT-*tree*. We will refer to them collectively as the set of OPT-*lists*, or just OPT for short. This representation of Q^* will allow us to compare the savings earned by the Weave Share solution Q to that of Q^*. Recall that m (respectively, m^*) is the number of trees in the final Weave Share plan Q (respectively, Q^*).

Note that by following this procedure, the OPT-lists may get fully populated before Weave Share finishes (if the final merges of WS are between existing multi-trees). And

```
initialize all m* OPT-lists (one list for each tree of Q*) to empty
{now consider the steps of Weave Share algorithm on the set of queries, one by one}
for each merge i of the Weave Share algorithm, i = 1 ... μ do
    if merge i is between two individual queries then
        let the two queries be called x and y
        add query x to the end of the OPT-list it belongs to
        add query y to the end of the OPT-list it belongs to
    else if merge i is between a multi-tree and a single query then
        add that query to the end of the OPT-list it belongs to
    else {merge i is between 2 multi-trees}
        the queries involved have already been added in a previous iteration
    end if
end for
```

conversely, Weave Share may finish before the OPT-lists get fully populated, if WS leaves many queries as stand-alone queries. In the latter case, the incomplete OPT-lists may be populated in an arbitrary order.

Construction of OPT-sequence The sought-after final sequence of merges leading to Q^* can be attained by considering the OPT-lists in arbitrary order and walking through each tree-list from right to left, merging one query at a time into its corresponding final OPT-tree. Recall that μ^* denotes the number of merges in OPT. In executing this OPT sequence of merges, we are effectively starting at the query plan N where there is no sharing—all queries are in their own individual execution trees—and proceeding step by step to the optimal query plan Q^*. After each merge, we are at an intermediary query plan, where some queries that OPT will eventually merge are still individual queries, and some of the final OPT-trees are not complete. We denote this sequence of intermediate query plans $(N = Q_0, Q_1, \ldots, Q_{\mu^*} = Q^*)$, numbered in the order specified by the procedure above, and we refer to it as the OPT-*sequence*. We will abuse this term and also use it to refer to the sequence of merges made by OPT in proceeding from $N = Q_0$ to Q^*.

Consider the change in cost between each adjacent pair of query plans in the OPT-sequence. Let us define for $j = 1 \ldots \mu^*$, each change in cost $r_j^* = C(Q_{j-1}) - C(Q_j)$. We now sort the r_j^*'s in non-increasing order and renumber them so that

$$r_1^* \geq r_2^* \geq r_3^* \geq \ldots \geq r_{\mu^*}^*. \tag{4}$$

A given WS merge (either between two individual queries, a query and a tree, or two trees) is said to *map to* a query in an OPT-list if, upon executing the merge, that query is added to an OPT-list in the above procedure. The following lemma is implicit in the construction procedure of the OPT-lists. For completeness we give a proof.

Lemma 2. *Each merge of Weave Share maps to at most two queries in the OPT-lists defined above. And each query in an OPT-list is only mapped to once.*

Proof. If at merge i, Weave Share merges two existing multi-trees, then merge i of Weave Share adds no new queries to the OPT-lists. If merge i involves one stand-alone

query (in the case of merging a stand-alone query into a multi-tree), then it adds one query to an OPT-list. If merge i involves two stand-alone queries then it adds two queries to one or two OPT-lists. Hence each Weave Share merge maps to at most 2 of OPT's merges. Each query is only added once because the procedure specifies that queries are added to an OPT-list only the first time they are "touched" by Weave Share: when they are still stand-alone queries.

Loosely speaking, the construction procedure of the OPT-sequence above accounts for each r_j^* using a WS merge. It effectively ensures that each WS merge precludes at most two of OPT's merges in the OPT-sequence defined above.

3.3 Proof of main theorem

We are now ready to prove Theorem 1. The idea of the proof is to break both OPT and Weave Share down into a sequence of merge-steps and then compare the savings achieved by Weave Share at each step to some corresponding savings achieved by OPT. This allows us to compare the total savings achieved by Weave Share with the total savings achieved by OPT.

Proof. First we handle a formality of bookkeeping. Recall that by assumption we have:

$$\mu \geq (1/2)\mu^* \tag{5}$$

Further recall that each merge of WS may map to 2 queries in OPT, which means WS may be making more merges after all of the queries in OPT have been mapped. Hence we simply define $r_j^* = 0$ for all $j = \mu^* + 1 \ldots 2\mu$.

We now proceed with our main argument. We denote the reductions in cost of each WS merge to be $r_1, \ldots, r_\mu$, indexed by the order of merges executed by the Weave Share algorithm. That is, r_i is the savings earned from the i^{th} merge of the Weave Share algorithm. The bulk of the proof will be dedicated to showing that for any $i = 1 \ldots \mu$, we have $2r_i \leq r_{2i-1}^*$. We begin with $i = 1$.

Savings at the first iteration of Weave Share Weave Share starts by merging two individual queries. The savings achieved at this step is at least half of the maximum individual savings achieved by OPT, i.e. $2r_1 \geq r_1^*$. Indeed, if r_1^* is achieved by merging an individual query into a multi-tree, the inequality follows from Lemma 1 in Section 3.1. If r_1^* is achieved by merging two stand-alone queries, then from the greedy nature of Weave Share it follows that $r_1 \geq r_1^*$. Note that by definition of the OPT-sequence, r_j^* was not achieved by merging two multi-trees, for any $j = 1 \ldots \mu^*$. Hence, we obtain $2r_1 \geq r_1^*$. This also implies $2r_1 \geq r_2^*$, due to the re-numbering in (4).

We refer the reader to Example 3 of the Appendix, which shows the argument for step $i = 2$ explicitly to help build intuition and make the following more concrete.

Savings at i^{th} iteration of Weave Share We now generalize the argument for savings r_i that is obtained at the i^{th} merge of Weave Share. We will demonstrate that $2r_i \geq r_{2i-1}^*$, and hence that also $2r_i \geq r_{2i}^*$ (since $r_{2i-1}^* \geq r_{2i}^*$). Note that if $2i-1 > \mu^*$ then the claim is trivially true, since then $r_{2i-1} = 0$. Hence we restrict our attention to the case that

$2i-1 \leq \mu^*$. Since Weave Share is greedy, r_i is at least the savings that we can get from any merge of two stand-alone queries that are still available just before the ith merge of Weave Share. So let us consider the quality of the merges that are still-available to WS at this juncture.

Recall the definition above of x_j and y_j for any r_j^* where $j \geq \mu^*$. At merge i of Weave Share, either:

1. x_{2i-1} and y_{2i-1} are still stand-alone queries, available to be merged by Weave Share, or
2. Weave Share has already used either x_{2i-1} or y_{2i-1} in one of its earlier $i-1$ merges. In this case another pair of queries (x_1 and y_1, x_2 and y_2,. . ., or x_{2i-2} and y_{2i-2}) are still available to be merged. We know this because by Lemma 2, a total of at most $2(i-1)$ OPT queries have been mapped by the first $i-1$ WS merges. By the pigeon hole principle, we must still have at least one pair of queries (x_1 and y_1, x_2 and y_2, …, or x_{j-1} and y_{j-1}) that are unmapped, i.e., stand-alone queries thus far untouched by WS.

Let us use x and y to denote the pair of queries that are still available to be merged of these possibilities. By Lemma 1, we know that merging x and y achieves at least as much reduction in cost as half of the reduction achieved when OPT merged x into the OPT-tree that x and y are a part of. Therefore, we know that

$$r(x,y) \geq \frac{1}{2}\min(r_1^*, \ldots, r_{2i-1}^*) \geq \frac{1}{2}(r_{2i-1}^*).$$

We also know by definition of WS that $r_i \geq r(x,y)$. Hence, for $i = 1 \ldots \mu$ we have:

$$\begin{aligned} &2r_i \geq r_{2i-1}^*, \text{ and} \\ &2r_i \geq r_{2i}^* \end{aligned} \tag{6}$$

We sum up the inequalities in (6) to get $2 \cdot 2(r_1 + r_2 + \ldots + r_\mu) \geq (r_1^* + r_2^* + r_3^* + \ldots + r_{2\mu}^*)$. And then, using inequality (5), we have $(r_1^* + r_2^* + r_3^* + \ldots + r_{2\mu}^*) \geq (r_1^* + r_2^* + r_3^* + \ldots + r_{\mu*}^*)$. Hence we obtain $4R(Q) \geq R(Q^*)$.

4 Summary and Open Problems

In this paper we have studied an important optimization problem for efficient sharing of ACQs for DSMSs. We analyzed a previously proposed greedy algorithm, Weave Share, which performed extremely well in an experimental study. We show that under some practical assumptions this algorithm guarantees a 4-approximation to the optimal cost-savings. Our analysis technique allowed us to elucidate some properties of the effect of sharing partial aggregates on total cost, and also required an exploration into the structural properties of an optimal solution. Several open questions remain. Determining whether there is a tighter analysis of WS, or an algorithm with a better approximation, are probably the two most immediate. Removing the assumptions required for proving our approximation may also be possible. An analysis of a follow-up algorithm for the problem that has been proposed, called Tri-Weave [8], would also be interesting.

References

1. D. J. Abadi, Y. Ahmad, M. Balazinska, U. C etintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.
2. Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 2003.
3. Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. In *SIGMOD*, 2003.
4. Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 434–443, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.
5. Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE TKDE*, 2007.
6. Shenoda Guirguis. *Scalable Processing of Multiple Aggregate Continuous Queries*. PhD thesis, University of Pittsburgh, 2011.
7. Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. Optimized processing of multiple aggregate continuous queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 1515–1524, New York, NY, USA, 2011. ACM.
8. Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.
9. Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A query processing engine for data streams. In *ICDE*, 2004.
10. Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
11. Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 2005.
12. Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.
13. K.V.M. Naidu, Rajeev Rastogi, Scott Satkin, and Anand Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.
14. Streambase, http://www.streambase.com, 2006.
15. System S, http://domino.research.ibm.com, 2008.

Appendix

In the Appendix we provide some helpful examples to further illustrate the concepts presented in the main paper. We also discuss exact solutions to two practical variations of our optimization problem that was defined in Section 2.3.

Examples

Example 1: trade-off between costs at the partial and final aggregation levels Consider an example instance [7] with two ACQs with ranges $r_1 = 12$, $r_2 = 10$ seconds and slides $s_1 = 9$, $s_2 = 6$ seconds. Note that the fragments of ACQ 1 are $g_1 = 3$, $g_2 = 6$, and the fragments of ACQ 2 are $g_1 = 4$, $g_2 = 2$. In unshared partial aggregation, the sub-aggregation operators of the first ACQ will produce 2 fragments every 9 seconds, while that of the second ACQ will produce 2 fragments every 6 seconds. We refer to the number of fragments generated per second as the *edge rate*, and denote it $E_1 = 0.22$, $E_2 = 0.33$ edges per second. With no sharing, the total final aggregation operations performed per second is $E_1 + E_2 = 0.55$. On the other hand, if shared partial aggregation is used, then we compose the two slides to form a composite slide of length $s' = 18$, with edges at times 3, 4, 6, 9, 10, 12, 16, and 18 seconds. This gives a combined edge rate of $E' = 8/18 = 0.44$. Since each ACQ has its own final aggregation operator, the total final aggregation operations performed per second in this case has increased to 0.88.

Example 2: Illustration for Observation 2 Consider three queries q_1, q_2, q_3. The total cost of the tree $[q_1, q_2, q_3]$ can be expressed as: $C([q_1, q_2, q_3]) = \lambda + E \cdot (\frac{r_1}{s_1} + \frac{r_2}{s_2} + \frac{r_3}{s_3})$, where r_i and s_i, $i = 1, 2, 3$ are the range and slide of query q_i. The values λ and $\frac{r_i}{s_i}$ stay constant and do not depend on the order of merges. The value E represents the edge rate of the tree. It is computed as the number of edges in a composite slide for the tree divided by length of the composite slide. The length of the composite slide is equal to the least common multiple of s_1, s_2, s_3 and clearly does not depend on the order of merges. The number of edges depends on how many prime divisors s_i and $r_i \mod s_i$ share, but not on the order of computation.

Example 3: Savings at the second iteration of Weave Share To build intuition regarding savings obtained at i^{th} iteration of Weave Share, here we explicitly write out how the savings obtained at the second iteration can be bounded. The savings achieved by the second merge of Weave Share is denoted by r_2. We will now demonstrate that $2r_2 \geq r_3^*$, and hence that $2r_2 \geq r_4^*$ (since $r_3^* \geq r_4^*$).

Since Weave Share is greedy, r_2 is at least the reduction that we can get from any still-available merge. So let us consider the quality of the merges that are still-available to WS before it makes its second merge to achieve a savings of r_2.

Recall the OPT-sequence $(Q_0, \ldots, Q_{\mu^*})$ that was created by following the procedure in Section 3.2. Further recall that r_j^* for $j \leq \mu^*$ is defined to be $C(Q_{j-1}) - C(Q_j)$, and this reduction in cost r_j^* was derived from merging a single query into an existing OPT-tree. We will refer to this query as x_j, and the last query that was added to the

same OPT-list before x_j will be referred to as y_j. I.e., x_j and y_j are adjacent in some OPT-list. We will use the terminology that x_j and y_j are *the queries of* r_j^*.

We know that currently, before the second merge of Weave Share, either:

- x_3 and y_3 are still individual, unmerged queries. They can potentially be merged by Weave Share
- or, if Weave Share's first merge (with savings r_1) already mapped to either x_3 or y_3, then either the two queries of r_1^* (x_1 and y_1) or the two queries of r_2^* (x_2 and y_2) are still unmapped, and hence available to be merged with each other.

This is true because each WS merge maps to at most 2 queries in OPT, and each query in OPT is mapped to at most once (Lemma 2). So at most 2 of OPT's merges are precluded by the first WS merge, leaving at least one pair of queries x_i and y_i, $i = 1, 2, 3$, available to be merged. Denote by x and y the two queries that are still available to be merged of these 3 pairs of possibilities.

By Lemma 1 we know that merging x and y achieves at least as much reduction in cost as half of the reduction achieved when OPT merged x into the OPT-tree that x and y are a part of. Hence, we can say

$$r(x, y) \geq \frac{1}{2} \min(r_1^*, r_2^*, r_3^*) \geq \frac{1}{2}(r_3^*).$$

We also know by the greedy nature of Weave Share that $r_2 \geq r(x, y)$. Taken together, we have: $2r_2 \geq r_3^*$.

Query pairing

Consider a restricted version of the optimization problem, in which each tree of the resulting query plan can have at most two queries in it. This version can be solved exactly and in polynomial time by reducing it to finding a maximum matching in a weighted undirected graph. Namely, we construct a graph in which each vertex corresponds to one input query. For every pair of vertices, an edge is added between them if sharing the corresponding queries leads to savings in processing cost. There are n vertices and $O(n^2)$ edges. The weight of the edge is the reduction in cost achieved by sharing the two queries.

An optimal query plan is found by maximizing the sum of savings achieved by individual merges; this is equivalent to solving a maximum weighted matching problem on the constructed graph: select a set of non-adjacent edges in a weighted graph so that the sum of their costs is maximized. After a graph corresponding to a set of queries is constructed, the maximum matching problem can be solved in $O(n^3)$ time by Gabow's algorithm [4].

Continuous query sharing

Another interesting variation is to consider an ordered sequence of ACQs and look for a query plan in which trees are formed from contiguous queries in order. For instance, if an "expiration date" is associated with each ACQ, the input sequence may be ordered

by this expiration date. This problem can be solved exactly and in polynomial time via a dynamic programming solution.

Let c_i be the value associated with each ACQ q_i that represents the minimum cost of the query plan for a subsequence of ACQs $\{q_i, \ldots q_n\}$. These values can be computed in right-to-left order: c_n is assigned the cost of executing q_n; for other values of i,

$$c_i = \min_{i \leq j \leq n} \{cost(q_i, \ldots, q_j) + c_{j+1}\},$$

where $cost(q_i, \ldots, q_j)$ denotes the cost of the tree containing a contiguous subsequence of queries $q_i, \ldots, q_j$.

The value c_1 associated with q_1 represents the cost of the least expensive query plan. To restore the actual partition of queries into trees, we associate a variable π_i with each query q_i and compute it in parallel with computing c_i, storing in π_i the index j, which yields the minimum cost in computation of c_i. That is, π_i holds the index of the "right boundary" of the tree that ACQ q_i is a part of.

Although this variation is somewhat restrictive in that it only allows contiguous subsets of ACQs to form trees, the advantage of this solution is that it adapts easily to query expiration: when, say, the first query expires and needs to be removed from the plan, the cost of the new solution is already stored in c_2 and the actual solution can be easily restored using π_2.