F1: Accelerating the Optimization of Aggregate Continuous Queries

Anatoli U. Shein, Panos K. Chrysanthis, Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
(aus, panos, labrinid)@cs.pitt.edu

ABSTRACT

Data Stream Management Systems performing on-line analytics rely on the efficient execution of large numbers of Aggregate Continuous Queries ($ACQs$). The state-of-the-art *WeaveShare* optimizer uses the *Weavability* concept in order to selectively combine $ACQs$ for partial aggregation and produce high quality execution plans. However, *WeaveShare* does not scale well with the number of $ACQs$. In this paper we propose a novel closed formula, $F1$, that accelerates *Weavability* calculations, and thus allows *WeaveShare* to achieve exceptional scalability in systems with heavy workloads. In general, $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple $ACQs$. We theoretically analyze the *Bit Set* approach currently used by *WeaveShare* and show that $F1$ is superior in both time and space complexities. We show that $F1$ performs $\sim 10^{62}$ times less operations compared to *Bit Set* to produce the same execution plan for the same input. We experimentally show that $F1$ executes up to 60,000 times faster and can handle 1,000,000 $ACQs$ in a setting where the limit for the current technique is 550.

1. INTRODUCTION

Nowadays more and more applications are becoming available to wider audiences, resulting in an increase in the amount of data produced. 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 be able to scale their services including on-line analytics.

On-line data analytics have gained momentum in many applications that need to ingest data fast and apply some form of computation. Data streams are part of this broader category of data management, where Data Stream Management Systems (*DSMS*) [1, 19, 3, 2, 14] have been proposed as suitable systems for handling large amounts of data, arriving with high velocity.

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, profit margin per stock, etc.) for different clients, each one with (possibly) different relaxation levels in terms of accuracy.

In *DSMS*, clients register their analytics queries on incoming data streams. Since these queries continuously aggregate streaming data to produce answers, they are called Aggregate Continuous Queries ($ACQs$). 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 often used to describe $ACQs$ are *range* (**r**) and *slide* (**s**) (sometimes also referred to as *window* and *shift* [9]). A slide denotes the period at which an ACQ updates its answer; 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. An ACQ requires the *DSMS* to keep state over time, while performing aggregations. Often, it is useful to run partial aggregations on the data while accumulating it, and then produce the answer by performing the final aggregation over the partial results [6, 11, 12, 13]. It is clear that the greater the range and the smaller the slide of the ACQ, the higher its cost is to maintain (memory) and process (CPU).

Problem Statement In this work we focus on environments where a large number of long-running $ACQs$ with different periodic properties (accuracies) are operating on the same data stream, calculating similar aggregate operations. Example of such an environment can be a *DSMS* deployed to *Cloud* Infrastructure, which results in a multitentant setting, where muliple $ACQs$ with a wide range of different periodic features are executed on the same hardware. Since the $ACQs$ are executed periodically (unlike one-shot queries), the opportunity to reduce the long-term overall processing costs by sharing final and partial results arises. For instance, assume that two $ACQs$ monitor an average stock value over the same data stream. Both have a slide of 3 sec. The first one has a range of 5 sec, whereas the second one has a range of 10 sec. In this setting, it is beneficial to keep calculating the results for the first query and combine the last two for generating the results for the second query. This should take place every 3 sec, which is the common slide among those $ACQs$. Partial results sharing is applicable for all matching aggregate operations, such as *max*, *count*, *sum*, *average*, etc., and for different but com-

CIKM'15, October 19–23, 2015, Melbourne, VIC, Australia

ACM 978-1-4503-3794-6/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2806416.2806450.

patible aggregate operations, for example *sum*, *count* and *average* can share results by treating *average* as *sum/count*.

Typically, the number of *ACQs* with similar aggregation types can be overwhelming in on-line systems [3]. Therefore, it is crucial to be able to make decisions quickly on combining different *ACQs* into execution plans that would benefit the system. Unfortunately, the former has been proven to be NP-hard [24], and currently only approximation algorithms can produce acceptable execution plans.

The state-of-the-art *WeaveShare* algorithm produces very high quality execution plans by utilizing the *Weavability* concept [8], which is used to decide which *ACQs* are similar enough to be combined. *WeaveShare* is theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem [4]. However, when we tried to implement it in a multiple-tenant *DSMS*, we observed that the current approach of calculating Weavability using *Bit Set* is very computationally expensive.

This motivated us to explore a more efficient algorithm to accelerate the calculation process in order to make the *WeaveShare* algorithm more scalable for systems with heavy workloads. Towards this, in this paper we propose a mathematical solution *Formula 1* (or $F1$), which reduces the number of operations needed to produce the efficient execution plan by $\sim 10^{62}$ times for the same set of 500 queries, and speeds up the plan generation time in our experiments by up to 60,000 times. $F1$ also eliminates concerns over the amount of system memory as it does not need to store any large data during its operation. In fact, $F1$ acceleration has enabled us to explore additional cost savings that can be achieved by utilizing the distributed nature of the *Cloud* Infrastructure by intelligently colocating *ACQs* on different computing nodes [17]. In general, $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple *ACQs*.

Contributions We make the following contributions:

- We propose a novel closed formula, $F1$, for calculating the number of overlapping partial result calculations between the *ACQs*, and show that $F1$ is applicable for all cases (with and without fragments).
- We theoretically evaluate the state-of-the-art *Weavability* calculation approach against $F1$ and provide a mathematical proof that $F1$ improves the time complexity from **at least** $\boldsymbol{e^n}$ to **at most** $\boldsymbol{2^n}$, and the space complexity correspondingly from **at least** $\boldsymbol{e^n}$ to **constant**.
- We experimentally evaluate $F1$ and show that we can achieve a speed increase of up to 60,000 times over the state-of-the-art techniques, which makes the processing of *ACQs* much more effective in an on-line *DSMS*.
- We show experimentally that $F1$ allows significantly better scalability in terms of the number of *ACQs*, input rate, and diversity of the *ACQ*'s time properties. We show that $F1$ is able to successfully process 1,000,000 *ACQs* when the limit of the current technique is 550.

Roadmap In the next section, we provide the background of our work. We introduce our new formula, $F1$, for the *Weavability* calculation and its additional optimization in Section 3. The complexity analysis on it is presented in Section 4. The evaluation platform and the experiments are discussed in Section 5. We summarize related work in Section 6 and conclude in Section 7.

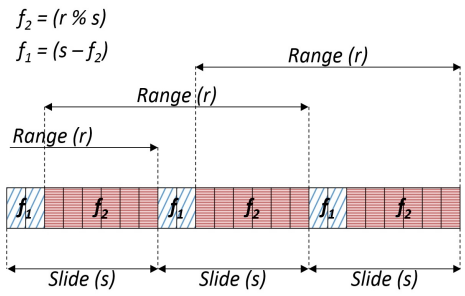

Figure 1: Paired Window Technique

2. BACKGROUND

In this paper, without a loss of generality, we target *ACQs* installed on the same data stream. Specifically, we scale the *WeaveShare* algorithm that has been shown to produce high quality execution plans of multiple *ACQs* by intelligently sharing partial aggregations.

Partial aggregation was proposed to improve the processing of *ACQs* [6, 11, 12, 13]. The idea behind partial aggregation is that we assemble the final answer from partial aggregates by performing a final aggregation over partials. For example, if an *ACQ* needs to calculate the maximum value over a specific time period, it can compute the maximum on each separate partition of streaming data, and then perform the final maximum operation on all obtained partial answers in order to get the result. Therefore, if the data set for the operation is split into n partitions, the system needs to perform n partial aggregations and one final aggregation to obtain the answer.

The question that immediately arises is how many partial aggregations are beneficial to perform before each final aggregation for a specific *ACQ*. This depends on the time properties of the *ACQ*.

- *The* ACQ *slide is greater or equal to the* ACQ *range.* In this case, partial aggregations are not beneficial. Only one aggregation is performed.
- *The* ACQ *slide is less than the* ACQ *range, and it divides the range evenly.* In this case it is beneficial to perform a partial aggregation every time period equal to the length of the slide.
- *The* ACQ *slide is less than the* ACQ *range, and it does not divide the range evenly.* In this case, it is beneficial to use the *Paired Window Technique* [11]. As can be seen from Figure 1, to use this approach one needs to calculate two fragment lengths $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$. The second fragment can be found by performing range *mod* slide, and the first fragment by subtracting the second fragment from the slide. Partial aggregations should be computed at periods of fragment one and fragment two interchangeably.

Shared Processing of ACQs Several shared processing schemes as well as multiple *ACQs* optimizers utilize the *Paired Window Technique* [11, 8]. 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

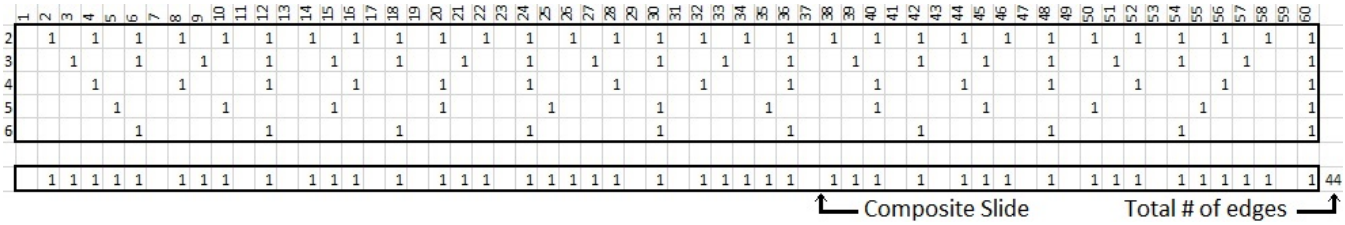

Figure 2: Marking edges produced by five different *ACQs* with NO fragments in the composite slide, represented by a Bit Set

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* then will 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.

The procedure to determine how many partial aggregations is needed after combining n *ACQs* using a *Bit Set* is formalized as follows:

- Find the length of the new combined (composite) slide, which is the *Least Common Multiple* (*LCM*) of all the slides of the 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*.
- 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*.

Weavability 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. Intuitively, two *ACQs* weave perfectly when their combined *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.

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

Note that 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 the total processing costs of the query plan. *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 reduce the total plan cost the most, into a single tree. *WeaveShare* produces a final execution plan when it cannot find a pair that would reduce the total plan cost further.

The most complex part of the calculation occurs when the system is scheduling each partial aggregation operation (*edge*) and is tracking these operations using a *Bit Set* as mentioned above. The size of the *Bit Set* increases rapidly if the *ACQs*' time properties differ. For each *ACQ* added to the execution tree, *WeaveShare* needs to traverse the whole *Bit Set* to make sure that all partial aggregations necessary for this *ACQ* are marked in a *Bit Set* for future execution. Since the size of a *Bit Set* increases exponentially with the increase of the input size, traversing it becomes prohibitively time-consuming as we show in Section 4. Additionally, the exponential increase of the size of the *Bit Set* puts a hard limit on a system's capabilities, based on the amount of memory available.

3. FORMULA 1 (F1)

In this section, we describe our new formula $F1$ that significantly speeds up the edge rate calculation in a composite slide. We target two scenarios for *ACQs* with matching or compatible aggregate operations: 1) when all *ACQ* slides are factors of their corresponding ranges, and 2) when some of the ranges are not multiples of their corresponding slides.

3.1 Case with *NO Fragments*

In the case when all of the ranges of the *ACQs* that are being installed onto the *DSMS* are divisible by their corresponding slides, we can store partial aggregates at every slide. For example, if we have an *ACQ* with a slide of 3 sec and a range of 9 sec, we can store partial results every 3 sec, and perform the final aggregations on the 3 last saved partial aggregations to get the answer for the last 9 sec. In order to calculate the edge rate in *WeaveShare* after *weaving* together n *ACQs*, we need a *Bit Set* of the length equal to the *LCM* of all n slides. At first, the *Bit Set* is populated with zeros. For each one of n *ACQs* we traverse the whole *Bit Set* and mark all bits whose indexes are divisible by the corresponding *ACQ*'s slide with ones. If the bit was already marked, the algorithm does nothing and just moves to the next bit.

Example 2 Consider five stock monitoring *ACQs* with the following slides: 2, 3, 4, 5, and 6. Their *LCM* is 60, therefore we need a *Bit Set* of size 60. First, we traverse the *Bit Set* and mark all indexes divisible by 2 (all even numbers up to 60). Now the *Bit Set* has 30 bits marked. Next we mark all indexes that are divisible by 3. The *Bit Set* already has 40 bits marked (10 overlapped with already marked ones). Next we mark all indexes that are divisible by 4. The *Bit Set* still has 40 bits marked since all of the bits we were trying to mark were already marked by a slide of 2. After repeating

the same for slides of 5 and 6, we calculate how many bits we have in our *Bit Set*, and the answer is 44. This method is illustrated in the Figure 2.

To accelerate this calculation process we propose the *Formula 1* (or $\boldsymbol{F1}$):

$$LCM_n \sum_{i=1}^{n} [(-1)^{i+1} G_1(n,i)] \tag{2}$$

Where $LCM_n = LCM(s_1, s_2, \ldots, s_n)$, and function $G_1(n,i)$ is a sum of the inversed LCMs of all possible groups of slides of size i from a set of size n. For example:

$$G_1(3,2) = \frac{1}{LCM(s_1,s_2)} + \frac{1}{LCM(s_1,s_3)} + \frac{1}{LCM(s_2,s_3)}$$

$F1$ can be expanded as follows:

$$LCM_n[G_1(n,1) - G_1(n,2) + \ldots \pm G_1(n,n-1) \mp G_1(n,n)] \tag{3}$$

Equation 3 is composed of an alternating series of function G_1 multiplied by the LCM_n. $LCM_n \cdot G_1(n,1)$ and represents the number of all edges produced by all *ACQs* and therefore includes all overlapping edges. The goal of the calculation is to count every edge only once, even if it overlaps multiple times in different *ACQs*. Therefore, the rest of the elements of the series will eliminate all of the overlapping edges from the current result. $LCM_n \cdot G_1(n,2)$ represents the number of edges that overlap in all different pairs of slides and after subtracting it, we get a smaller number than the number we are looking for, because there are potentially some edges where more than two slides overlap at the same time. For example, if slides a, b, and c overlap at some specific edge e, we add it three times: for pairs (a,b), (a,c), and (b,c). The following element $LCM_n \cdot G_1(n,3)$ compensates for these cases by adding back all edges that overlap in each set of three slides. After adding it, we have again a larger number than the sought-after number, because we might have four or more slides overlapping at the same edge. Therefore, each element compensates for the previous ones' inaccuracies up to the point when we add/subtract the final edge of the composite slide, which clearly occurs only once, since $LCM_n \cdot G_1(n,n) = \frac{LCM_n}{LCM_n} = 1$. The last added/subtracted edge has an index equal to the LCM_n.

Equation 3 is an alternating series and we know in advance that the number of elements is always equal to the number of *ACQs* in the execution tree and is a finite number. Therefore, by definition, the sequence always converges.

The following is an example of using $F1$ to calculate the number of edges:

Example 3 Consider the same set of stock monitoring *ACQs* as we had in Example 1: slides are $2, 3, 4, 5$, and 6. As a first step of our algorithm we calculate the LCM_n of the whole set of slides. $LCM_n = LCM(2,3,4,5,6) = 60$. Next we substitute our values into Equation 11:

$$60 \cdot G_1(5,1) - 60 \cdot G_1(5,2) + 60 \cdot G_1(5,3) - 60 \cdot G_1(5,4) + 1$$

Every element is expanded as shown above. For example, the expansion of element $60 \cdot G_1(5,2)$ is as follows. (Note that LCM_{ab} denotes $LCM(a,b)$).

$$60 \cdot G_1(5,2) = \frac{60}{LCM_{23}} + \frac{60}{LCM_{24}} + \frac{60}{LCM_{25}} + \\ + \frac{60}{LCM_{26}} + \frac{60}{LCM_{34}} + \frac{60}{LCM_{35}} + \frac{60}{LCM_{36}} + \\ + \frac{60}{LCM_{45}} + \frac{60}{LCM_{46}} + \frac{60}{LCM_{56}} = 70$$

Finally we have: $87 - 70 + 36 - 10 + 1 = 44$
This answer matches the solution from Example 1.

Notice that the elements of the alternating series are interchangeably increasing and decreasing the solution as we approach the end of the calculation. For 20 different *ACQs*, the calculation of overlapping edges using $F1$ consists of 20 addition operations, causing the total number to change as depicted in Figure 3.

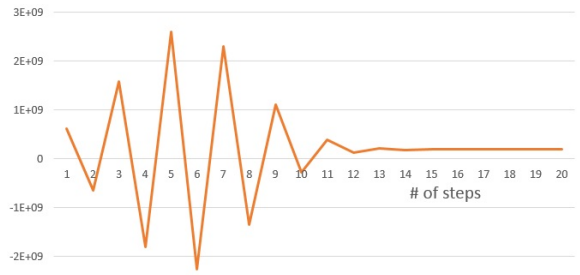

Figure 3: $F1$ converging to the solution for 20 *ACQs* in 20 steps

3.2 Case *WITH Fragments*

In case some of the ranges of the *ACQs* that are being installed onto the *DSMS* are not divisible by their corresponding slides, according to the *Paired Window* approach, the slides should be broken into fragments. This enables us to store partial aggregates for every fragment. For example, if we have an *ACQ* with slide 5 sec and range 7 sec, the slide is split into two fragments: $f_2 = 7 \pmod 5 = 2$ and $f_1 = 5 - 2 = 3$. Now we can store partial results for first 3 sec, then for following 2 sec, then again for the following 3 sec and so on.

In the original *WeaveShare* [8], in order to calculate the edge rate after *weaving* together n *ACQs* with fragments, we again need to work with a *Bit Set* of the length equal to the *LCM* of all n slides. The Bit Set is pre-populated with zeros again. For each *ACQ* we traverse the whole Bit Set and mark bits corresponding to the times when partial aggregations will happen with ones. If the bit was already marked, the algorithm does nothing and just moves to the next location.

Example 4 Consider four stock monitoring *ACQs* with the following slides: $3, 4, 6$, and 9. *ACQs* with slides of 4 and 6 consist of fragments $(3,1)$ and $(2,4)$ respectively. *ACQs* 3 and 9 do not have fragments. The overall *LCM* of all slides together is 36, therefore we need a *Bit Set* of size 36. First, we traverse the *Bit Set* and mark all indexes that are divisible by 3 for the *ACQ* with a slide of 3 and no fragments. Now the *Bit Set* has 12 bits marked. Next consider an *ACQ* with a slide of 4 and fragments $(3,1)$. We traverse the *Bit Set* by starting from 0 and adding fragment 3 followed by fragment 1 repeatedly. Thus, the *Bit Set* will be marked at

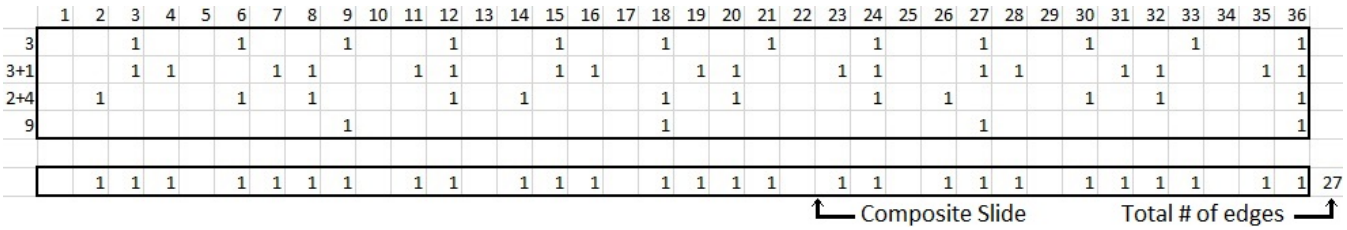

Figure 4: Marking edges produced by four different *ACQs* WITH fragments in the composite slide, represented by a Bit Set

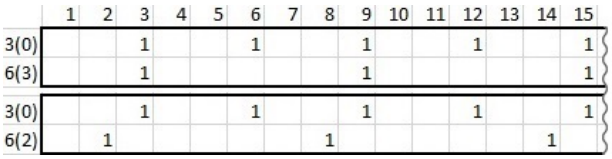

Figure 5: (slide 3, shift 0) and (slide 6, shift 3) **DO overlap**, but (slide 3, shift 0) and (slide 6, shift 2) **DO NOT**

indexes $3, 4, 7, 8, 11, 12$, etc. Now there are 24 marked bits. Next we continue to an *ACQ* with a slide of 6 and fragments $(2, 4)$. Again, we start at 0 and by adding 2 and 4 repeatedly we mark the following bits: $2, 6, 8, 12, 14$, etc., marking 27 bits in total. For the last *ACQ* with a slide of 9 and no fragments, we traverse the *Bit Set* at increments of size 9 and mark each 9th bit with one. The total number of set bits stays 27, because the last *ACQ* did not add any new bits, therefore our answer is 27. This method is illustrated in Figure 4.

To **generalize** $F1$ for both cases (if we do have *ACQs* with fragments and if we do not) we introduce the notion of *shifts*. Each *ACQ* that does not have fragments has a *shift* of zero. Each *ACQ* that does have fragments must be presented as two *ACQs* with the same slides, but different *shifts*. First one has a shift of zero, and the second one has a *shift* equal to the first fragment of the original *ACQ*.

When counting overlapping edges of *ACQs*, and when at least one of their shifts is not zero, we can encounter two different cases:

- *ACQs* overlap, and the number of common edges is the same, as it would be if all of the *ACQs*' shifts were zeros.
- *ACQs* do not overlap at all. Since the shifts are not compatible, the number of common edges is zero.

Example 5 Assume two *ACQs* with slides of 3 and 6. If their corresponding shifts are 0 and 3, there is an overlapping edge every 6 time units. However, if the corresponding shifts are 0 and 2, there are no overlapping edges. This is illustrated in Figure 5.

To decide whether two *ACQs* q_1 and q_2 (if at least one of them has non-zero shift) will overlap, we propose the following **Overlap Check Formula** based on *GCD* (*Greatest Common Divisor*):

$$|q_1.shift - q_2.shift| \mod GCD(q_1.slide, q_2.slide) \quad (4)$$

- If the *Overlap Check Formula* resolves to *zero* then the *ACQs* **DO** overlap
- *Otherwise* the *ACQs* **DO NOT** overlap

Proof of the **Overlap Check Formula** (by contradiction) Assume that we have two *ACQs* q_1 and q_2, with corresponding slides s_1 and s_2, and shift difference h. Assume further that $h \bmod GCD(s_1, s_2) \neq 0$ and (for the sake of contradiction) the *ACQs DO* overlap. Let us denote all edges produced by q_1 as $\{e_{1-1}, e_{1-2}, \ldots, e_{1-n}\}$, and edges of q_2 as $\{e_{2-1}, e_{2-2}, \ldots, e_{2-n}\}$. Let us first look at the two *ACQs* separately. Since every edge produced by q_1 is divisible by s_1, and every edge produced by q_2 is divisible by s_2, and both s_1 and s_2 are divisible by $GCD(s_1, s_2)$ (by definition of GCD), every edge produced by q_1 and q_2 is divisible by $GCD(s_1, s_2)$. Therefore, all edges that are not divisible by $GCD(s_1, s_2)$ cannot possibly overlap any of the edges produced by either q_1 or q_2. Without loss of generality, let us consider q_2 from the standpoint of q_1. Then, all edges of q_2 are shifted by h with respect to edges of q_1, and they can be written as follows: $\{e_{2-1} + h, e_{2-2} + h, \ldots, e_{2-n} + h\}$. These edges should be divisible by $GCD(s_1, s_2)$ in order for them to overlap the edges of q_1: $\{e_{1-1}, e_{1-2}, \ldots, e_{1-n}\}$. We know that the edges $\{e_{2-1}, e_{2-2}, \ldots, e_{2-n}\}$ are divisible by $GCD(s_1, s_2)$. However, by the initial assumption, the shift h that is being added to them is not divisible by $GCD(s_1, s_2)$. Thus, edges $\{e_{2-1} + h, e_{2-2} + h, \ldots, e_{2-n} + h\}$ cannot possibly be divisible by $GCD(s_1, s_2)$. Therefore, none of the edges of q_2 can possibly overlap with the edges of q_1. In the case that h would actually be divisible by $GCD(s_1, s_2)$, all shifted edges of q_2 that are divisible by s_1 would overlap with the edges of q_1. However, the initial assumption states that h is not divisible by $GCD(s_1, s_2)$, which leads us to the conclusion that the *ACQs* q_1 and q_2 do not overlap, which is a contradiction. Hence, the initial formula is correct. ■

Next we generalize our formula $F1$ for use in cases when *ACQs* have fragments, and cases when none of the *ACQs* have fragments. The **general F1** is:

$$LCM_n \sum_{i=1}^{n} [(-1)^{i+1} G_2(n, i)] \quad (5)$$

Where again $LCM_n = LCM(s_1, s_2, \ldots, s_n)$, and function G_2 is the same as function G_1, however all elements produced by G_2 have to be checked with the *Overlap Check Formula* for redundancy as described below. Prior to using this formula, for each *ACQ* that has fragments, we create two new *ACQs*: one of them has a shift of zero, another one has a shift equal to the first fragment of the original *ACQ*. All new *ACQs* are added back to the set of the original *ACQs* replacing the originals. To calculate each G_2 we find all possible groups of size x from the new set of *ACQs* just like in the case with no fragments. Some of these groups are redundant because they do not have overlapping edges (because of the shifts). To remove all redundant groups, we check all possible pairs within each group using the *Overlap*

Check Formula, and if any of the pairs return a non-zero value, then the whole group is discarded. Otherwise, G_2 is calculated and used the same way as in the case with *NO* fragments. The generalized formula $F1$ still converges, which can be proven using the same strategy as in the case with no fragments.

Equation 5 expands into an alternating series likewise:

$$LCM_n[G_2(n,1) - G_2(n,2) + \ldots \pm G_2(n,n-1) \mp G_2(n,n)] \quad (6)$$

We show how Equation 6 works with the following example.

Example 6 Assume the same set of stock monitoring *ACQs* as in Example 4: slides are 3, 4, 6, and 9, and *ACQs* with slides of 4 and 6 consist of fragments $(3,1)$ and $(2,4)$ respectively. As a first step of our algorithm we calculate the LCM_n of the whole set of slides. $LCM_n = LCM(3,4,6,9) = 36$. Next we replace the *ACQs* that have fragments with the *ACQs* that have corresponding shifts. In our set we now have two *ACQs* with a slide of 4 (shifts 0 and 3), and two *ACQs* with a slide of 6 (shift 0 and shift 2). The rest of the *ACQs* stay the same. We can substitute our values into the generalized formula $F1$:

$$36 \cdot G_2(6,1) - 36 \cdot G_1(6,2) + 36 \cdot G_1(6,3) - 36 \cdot G_1(6,4)$$

The calculation is almost identical to the case with no fragments, except every group produced by G_2 has to be checked with the *Overlap Check Formula* to see if it is redundant or not. For example, the expansion of the second group is shown below. Note that $3_0 4_3$ denotes a group of *ACQs* with slides of 3 and 4 and shifts of 0 and 3, respectively. The fractions that have been crossed out did not pass the test with the *Overlap Check Formula*.

$$36 \cdot G_1(6,2) = \frac{36}{LCM_{3_0 4_3}} + \cancel{\frac{36}{LCM_{3_0 6_2}}} + \frac{36}{LCM_{3_0 9_0}} + \frac{36}{LCM_{3_0 4_0}} + \frac{36}{LCM_{3_0 6_0}} + \cancel{\frac{36}{LCM_{4_3 6_2}}} + \frac{36}{LCM_{4_3 9_0}} + \cancel{\frac{36}{LCM_{4_3 4_0}}} + \cancel{\frac{36}{LCM_{4_3 6_0}}} + \cancel{\frac{36}{LCM_{6_2 9_0}}} + \frac{36}{LCM_{6_2 4_0}} + \cancel{\frac{36}{LCM_{6_2 6_0}}} + \frac{36}{LCM_{9_0 4_0}} + \frac{36}{LCM_{9_0 6_0}} + \frac{36}{LCM_{4_0 6_0}} = 26$$

Finally we have: $46 - 26 + 8 - 1 = 27$
This solution matches the solution from Example 4.

3.3 *F1 Optimization*

Since we are using the Euclidean GCD algorithm for all of our LCM calculations, we found that we can achieve a significant additional speed up by utilizing the technique of memorization. We adopted this technique by preloading a table of GCDs into main memory before the execution begins. If the user is willing to allocate b bytes of memory to store the GCD table and each GCD takes g bytes of memory, we can store in memory GCDs of all the possible pairs of numbers up to $\sqrt{2b/g}$. In our implementation we are using 8 byte numbers of the Long type for calculations, so if we want to allocate 4 GB of main memory to store $GCDs$, we can fit GCDs of all the pairs of numbers up to $32,768$. If we calculate the GCD for numbers that are larger than the above limit, the GCD table still save us some time by taking advantage of the recursive nature of the Euclidean algorithm. The effects of the optimization are shown in Section 5.

4. COMPLEXITY ANALYSIS

In this section, we calculate the difference between the complexities of *Bit Set* calculation and our $F1$ method.

Time Complexities To compare the time complexities we start by identifying the initial calculations needed by both algorithms. We denote the number of *ACQs* as n, and the max slide as max. The following steps need to be done at the beginning of both algorithms.

- Remove all duplicate slides, since the same slides produce the same edges, and we do not want to repeat the same calculation for every duplicate. This is done by sorting slides with duplicate removal in $n \cdot logn$ time.
- Precalculate the LCM_n, which is the LCM of all slides and store it in the main memory. This operation takes $(n-1) \cdot log(max)$ at worst, since we need to perform LCM operation pairwise $n-1$ times, and each $LCM(a,b)$ needs at worst $log(min(a,b))$ operations [21].
- Remove all slides that are multiples of other slides included in the set. We do this because all edges produced by such slides are already produced by their factors. This can be done in $n \cdot (n-1)/2$ operations since our slide set is sorted, and for each subsequent slide we need to do a number of comparisons that equals the number of comparisons performed by the previous slide minus one.

Therefore, precalculation takes $n \cdot logn + (n-1) \cdot log(max) + n \cdot (n-1)/2$ operations, however since it is performed by both algorithms, we can ignore it for the matter of comparison.

After completing the initial computation, we now have LCM_n stored in main memory, and a set of slides which does not contain any duplicates or multiples. Therefore, the set can now only have either prime numbers or numbers for which their multiples do not appear in the set.

To better illustrate differences in complexity we utilize two different sets of slides:

- Working Set (S_w) is a set of slides that includes only prime numbers and numbers that do not have their multiples in this set. This is a set produced after the initial preparation and it is used in real working scenarios. An example of a valid working set: $S_w = 3, 5, 8, 14, \ldots, max$. $|Sw| = n$, and the maximum element is denoted as max.
- Auxiliary set (S_a) is a set of slides, that consists of sequential numbers. $Sa = 1, 2, \ldots, n$. $|Sa| = n$, and the maximum value max equals n. Note that this set contains all natural numbers from one to n, including multiples of other numbers from this set. It is still a valid set for our computation, and can be obtained by skipping the preliminary optimization that removes all multiples.

Next we show that the lower bound of the *Bit Set* calculation is higher than the upper bound of the $F1$ computation.

The complexity of the *Bit Set* calculation is:

$$\sum_{i=1}^{n} \frac{LCM_n}{s_i} \quad (7)$$

Where $LCM_n = LCM(s_1, s_2, \ldots, s_n)$. The complexity holds since for each of n *ACQs* we would need to traverse the whole *Bit Set*, whose length is equal to the LCM of all slides of all *ACQs*, with a step equal to each *ACQ*'s slide. We can expand the Equation 7 to the following:

$$LCM_n \cdot (\frac{1}{s_1} + \frac{1}{s_2} + \ldots + \frac{1}{s_n}) \quad (8)$$

First, we perform complexity analysis using the Auxiliary set S_a. Let us focus on the first part of the product in Equation 8: LCM_n. We know that the LCM of all numbers in this set is the product of the highest prime powers occurring in the set. The *log* of the LCM is therefore the sum of the *logs* of the prime powers in the set:

$$log(LCM_n) = \sum_{i=2}^{n} f(i) \tag{9}$$

$$where f(i) = \begin{cases} log(p) & if\, i = p^m,\, where\, m \geq 1 \,\&\, p\, is\, prime \\ 0 & Otherwise \end{cases}$$

This sum has significance in the *Prime Number Theorem* and it is well known to be asymptotically equal to e^n [23].

Next we focus on the second part of the product in Equation 8: $(\frac{1}{s_1} + \frac{1}{s_2} + \ldots + \frac{1}{s_n})$. Since our set only has sequential numbers from one to n, this part will look like: $(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n})$, which is a classic example of a diverging harmonic series $\sum_{n=1}^{\infty} \frac{1}{n}$. For any n, this series can be calculated as follows: $\sum_{n=1}^{\infty} \frac{1}{n} = ln(k) + \gamma + \epsilon_k$, where γ is the Euler-Mascheroni constant ($\gamma \approx 0.577$) and $\epsilon_k \sim \frac{1}{2k}$, which approaches zero as k goes to infinity[22]. For our purposes, since γ is a constant and ϵ_k is negligible, they are ignored. Also, we can say that time complexity $e^n \cdot ln(n)$ is asymptotically equal to e^n, therefore we can assume asymptotical time complexity of the *Bit Set* computation for set S_a is e^n.

When we use set S_w, the time complexity for LCM_n is larger than when using set S_a. This is true because we replace n sequential numbers that start from 1 with n non-duplicate primes and non-multiples, which are larger and have a larger total LCM. The time complexity for the part $\sum_{i=1}^{n} \frac{1}{s_i}$ becomes smaller, because we are increasing numbers in the denominator, however it is still insignificant, because even in the worst case we can lower bound it with $ln(max) - ln(max - n)$. Thus, the time complexity of the *Bit Set* computation for the working set S_w is at least e^n.

To calculate the complexity of $F1$, we need to determine the number of operations that need to be performed based on the size of the input. First, we know that the number of elements in our alternating series is equal to the number of $ACQs$ in the set. Let us take Equation 3 and expand LCM_n into the parentheses:

$$\begin{aligned} &LCM_n \cdot G_1(n,1) - LCM_n \cdot G_1(n,2) + \ldots \\ &\pm LCM_n \cdot G_1(n,n-1) \mp LCM_n \cdot G_1(n,n) \end{aligned} \tag{10}$$

As we previously mentioned, $LCM_n \cdot G_1(n,n) = 1$, therefore:

$$\begin{aligned} &LCM_n \cdot G_1(n,1) - LCM_n \cdot G_1(n,2) + \ldots \\ &\pm\, LCM_n \cdot G_1(n,n-1) \mp 1 \end{aligned} \tag{11}$$

To determine how many groups will be produced by each one of these elements we use binomial coefficients. Each element of type $LCM_n \cdot G_1(n,k)$ therefore produces $\binom{n}{k}$ of distinct k-element groups of type $\frac{LCM_n}{LCM_k}$, where $LCM_k = LCM(s_1, s_2, \ldots s_k)$. Therefore, the total number of all of these groups is: $\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n-1} + \binom{n}{n}$. By the additive property of binomial coefficients, this sum equals $2^n - 1$. Next we determine how many calculations are performed in each group. The numerator of all groups is LCM_n and since it is kept in main memory, we do not need to recalculate it every time. The denominator is LCM_k, and it is determined by calculating the LCM of the first two elements, and then iteratively calculating the $LCMs$ of the resulting number with the rest of the elements in the group. Therefore, for each group we need to perform $k-1$ LCM calculations, and one calculation to add the group to the total number, which makes k calculations total. Since each group with k elements needs k calculations, the total number of calculations needed for all groups becomes: $1\binom{n}{1} + 2\binom{n}{2} + 3\binom{n}{3} + \ldots + (n-1)\binom{n}{n-1} + n\binom{n}{n}$. This resolves to $2^{n-1} \cdot n$, which can be calculated by taking the generalization of binomial series: $(1+x)^a = \sum_{k=0}^{\infty} \binom{a}{k} \cdot x^k$ and differentiating it with respect to x and then substituting $x = 1$ [20]. Due to the use of the Euclidean algorithm to calculate the LCM, the complexity of each LCM calculation is $log(min(a,b))$ at most [21]. Therefore, at worst $F1$ has a time complexity of $2^{n-1} \cdot n \cdot log(max)$, which asymptotically equals 2^n.

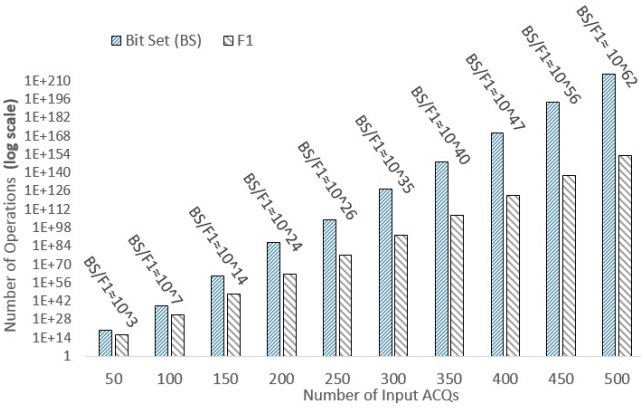

Figure 6: Number of operations needed by Bit Set and $F1$ for plan generation. Top labels show BitSet/$F1$ ratio

Thus, we have determined that the *Bit Set* calculation has a time complexity of **at least e^n**, and $F1$ has a time complexity of **at worst 2^n**. Clearly, when n goes to infinity, it is increasingly beneficial to use $F1$ versus *Bit Set*.

Additionally, since we have calculated the formulas for determining the exact number of operations done by both *Bit Set* and $F1$, we can compare the increase in the amount of operations performed by *Bit Set* and $F1$ with the increase of the number of input $ACQs$. The comparison is shown in Figure 6. $ACQs$ for this comparison were sequentially drawn from the Auxiliary set (S_a) introduced above. Note that since the difference between *Bit Set* and $F1$ operation numbers is drastic and grows exponentially we had to use a logarithmic scale to still see the operations of $F1$. This comparison shows that $F1$ is much more scalable than *Bit Set* in terms of the number of operations required.

Space Complexities The space complexity of the *Bit Set* calculation is LCM_n, since we have already shown that the *Bit Set* grows at the rate of e^n. The space complexity of the $F1$ calculation is $O(1)$ (*constant*) since it does not require storing edges. Edge overlaps are calculated strictly mathematically. Since $F1$ expands into a sum, we only need to keep one number in memory, which is increased or decreased by the elements of the alternating series sequentially. The improvement in space complexity is extremely important for the *WeaveShare* algorithm, since the leading cause of its failures with large workloads is "out of memory" errors.

5. EXPERIMENTAL EVALUATION

In this section, we summarize the results of our experimental evaluation of the scalability of $F1$ in terms of the size of the input set of the $ACQs$, the diversity of their time properties, and the input rate of the data stream.

5.1 Experimental Testbed

In order to show the significance of our *Weavability* calculation optimization we built an experimental platform in Java. Specifically, we implemented the *WeaveShare* optimizer as described in [8] with different options for calculating *Weavability*. 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 get a more detailed sensitivity analysis of the optimizer's performance. Moreover, it allows us to target possible real-life scenarios and analyze them.

Our system's **experimental parameters** are:

[***Algorithm***] specifies which technique is used for *Weavability* calculations. The available techniques are: (a) Bit Set (BS), (b) Formula 1 ($F1$), and (c) Formula 1 + Optimization ($F1 + Opt$). The $F1 + Opt$ technique uses a 4 GB table for keeping GCDs in main memory.

[$\boldsymbol{Q_{num}}$] Number of $ACQs$. We assume that all $ACQs$ are installed on the same data stream and their aggregate functions allow them to share partial aggregations among them. The actual function does not have any effect on performance other than the ability to share partial aggregations.

[$\boldsymbol{S_{max}}$] Maximum slide length, which provides an upper bound on how large slides of our $ACQs$ can be. The minimum slide allowed by the system always equals one.

[$\boldsymbol{\lambda}$] The input rate, which describes how fast tuples arrive through the input stream in our system.

[$\boldsymbol{Z_{skew}}$] Zipf distribution skew, which depicts the popularity of each slide length in the final set of $ACQs$. A Zipf skew of zero produces uniform distribution, and a greater Zipf skew is skewed towards large slides (for more realistic examples).

[$\boldsymbol{\Omega_{max}}$] Maximum overlap factor, 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.

[***Gen***] Generator type, which defines whether our workload is normal (Nrm) and includes any slides, or diverse (Div) and includes only prime slides.

5.2 Experimental Results

To test the scalability of our approaches $F1$ and $F1 + Opt$ versus BS in terms of the parameters Q_{num}, S_{max}, λ, Z_{skew}, and Ω_{max}, we ran five experiments, where we varied each one of these parameters while keeping the rest of them fixed. The parameters were selected separately for each experiment in a way that would highlight the differences in the scalabilities of the three approaches the best. The experimental parameters are specified in the Table 1.

All results are taken as averages of running each experiment five times. Please note that since $F1$ and $F1 + Opt$ showed to have significantly smaller runtimes compared to BS, we had to use **logarithmic scale** to be able to display all techniques' performances in the same graphs.

We ran all our experiments on a dual Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz server with 96 GB of RAM.

Exp 1: Number of ACQs Scalability (Figure 7)
In this test we varied the Q_{num} from 100 to 1,000,000.

Table 1: Experiment Parameters

#	Q_{num}	S_{max}	λ	Z_{skew}	Ω	*Gen*
1	*100 - 1M*	1K	0.002	0.5	10	*Nrm*
2	1K	*100-10*K	1	3	100	*Div*
3	1K	100	*100-1M*	1	1K	*Nrm*
4	50	500	0.1	*0-100*	100	*Nrm*
5	1K	600	1	3	*100-1*	*Div*

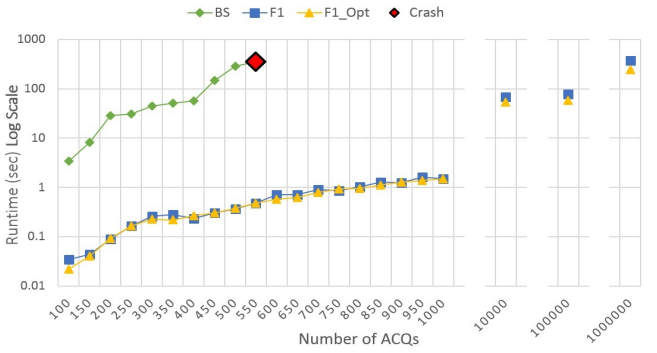

Figure 7: **Exp 1** Scalability of the number of ACQs

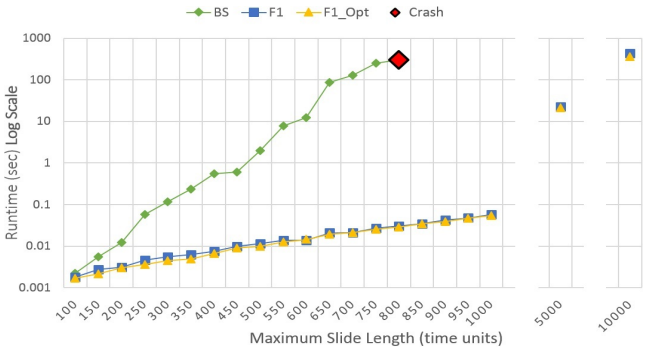

Figure 8: **Exp 2** Scalability of the maximum slide length

Clearly, increasing the Q_{num} also increases the amount of required calculations, causing higher runtimes for all three algorithms. The results are depicted in Figure 7. The *Bit Set* approach did not finish execution, because after we crossed Q_{num} of 550 it started running out of memory (on a 96GB RAM machine) and eventually crashed (on all runs). Otherwise, the growth rates of these techniques are similar to what we expected from the theoretical analysis of the time complexities of their underlying algorithms. The statistics show that our techniques' runtimes are on average 350 times faster than runtimes of BS with a maximum of 790 times, and our techniques are able to scale up to 1,000,000 $ACQs$ on this setting without running out of memory. Also, the $F1 + Opt$ plan outperformed the $F1$ plan by approximately 28% on average, validating our optimization expectations.

Exp 2: Max Slide Scalability (Figure 8)
In this test we varied the S_{max} from 100 to 10,000. Similarly to Exp 1, increasing the S_{max} also increases the amount of required calculations. This happens because with a higher max slide parameter, the generated $ACQs$ have longer slides, which results in higher $LCMs$ and fewer overlapping edges. In Figure 8 we see that the BS approach did not finish execution again after we crossed S_{max} of 800 because of the

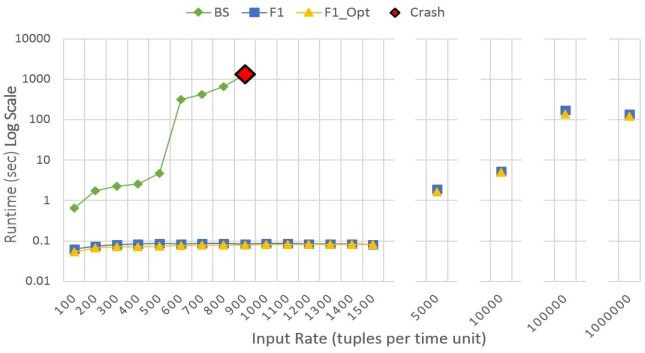

Figure 9: **Exp 3** Scalability of the input rate

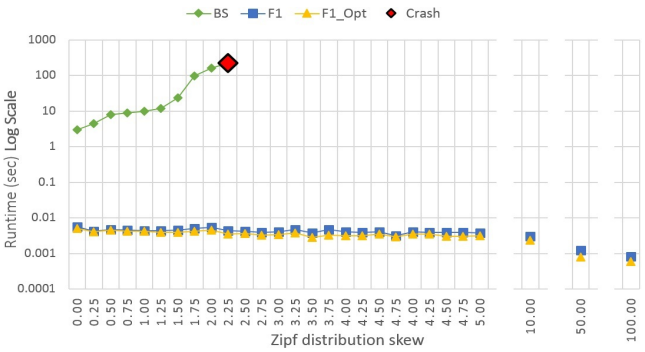

Figure 10: **Exp 4** Sensitivity to the zipf distribution skew

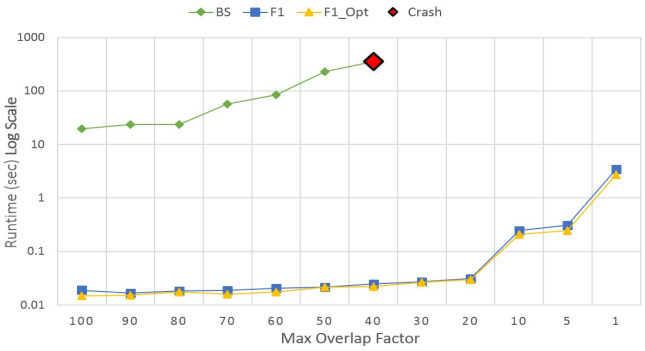

Figure 11: **Exp 5** Sensitivity to the maximum overlap factor

"out of memory" error. The growth rates of our techniques however are again similar to what we expected from our theoretical analysis of their time complexities. Our proposed techniques' runtimes in this experiment are on average 2,200 times faster than runtimes of BS with a maximum of 10,000 times, and our techniques are able to scale up to the S_{max} of 10,000 $ACQs$ on this setting and finish the plan generation successfully. The $F1 + Opt$ plan outperformed the $F1$ plan by an average of 18%.

Exp 3: Input Rate Scalability (Figure 9)
In this test we varied λ from 100 to 100,000. Increasing λ also increases the amount of required calculations because with higher input rates, according to the Equation 1, it becomes more beneficial to combine more execution trees. This forces *WeaveShare* to combine some trees with different time properties that would not have been combined if the input rate was lower. Thus, increasing λ leads to higher LCMs and higher runtimes. The average number of execution trees formed at the end of the plan generation is 71 when $\lambda = 100$, 29 when $\lambda = 400$, 15 when $\lambda = 900$, 4 when $\lambda = 10,000$ and 1 when $\lambda = 100,000$ or 1,000,000. The fact that at some λ all trees get merged into one explains why runtimes stop increasing after this λ. In this setting it happened at $\lambda = 100,000$ (see Figure 9). In this experiment the BS approach crashed when λ reached 900, and the average number of trees at that point was 15. Our approaches demonstrated good scalability again and were able to increase input rate to the point where all trees are *Weaved* into one. On average our techniques ran 3,800 times faster than BS with a maximum of 16,000. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 19%.

Exp 4: Slide Skew Sensitivity (Figure 10)
In this test we varied the Z_{skew} from 0 to 100. This experiment is similar to the max slide scalability experiment, because in both experiments we are gradually increasing the amount of $ACQs$ with large slides and therefore increasing the amount of required calculations. The difference is that, when skewing all slides drawn from the same set to the larger side, at some point they start repeating, which then reduces the amount of the required calculations. In our experiment (see Figure 10) we first observe the initial increase in the amount of computation, which leads the BS approach to crash with an "out of memory" error (at $Z_{skew} = 2.25$), and then we see gradual decrease in computation, because there are many repeating slides in the input set. In this setting our proposed techniques' runtimes are on average 14,000 times faster than runtimes of BS with a maximum of 60,000 times, and our techniques are able to scale up to the Z_{skew} of 100 and finish the plan generation successfully. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 18% again.

Exp 5: Overlap Factor Sensitivity (Figure 11)
In this test we varied the Ω from 100 to 1. We did it in reverse order since its value is inversely proportional to the amount of computation required to generate an execution plan using *WeaveShare*. Based on Equation 1 we can see that smaller Ωs benefit the total cost if their corresponding $ACQs$ are combined to fewer execution trees, which causes *WeaveShare* to *Weave* more trees with different time properties together. In our experiment (see Figure 11) the BS approach crashed when Ω reached 40. Our approaches again demonstrated good scalability and were able to finish the plan generation successfully even with the minimum value of $\Omega = 1$. On average our techniques ran 5,600 times faster than BS with a maximum of 16,000. The $F1 + Opt$ plan outperformed the $F1$ plan by the average of 26%.

Experimental Results Summary Clearly, the above experimental results show that our techniques $F1$ and $F1{+}Opt$ deliver the best performance in terms of plan generation runtimes and scalability, while producing same high quality execution plans as the original *WeaveShare* optimizer. The results of our experiments are summarized in Table 2.

6. RELATED WORK

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 pro-

Table 2: Experimental Results

Experiment		Best Achieved		Runtime: $BS/F1$		$F1$ vs
#	Param	BS	$F1$	Avg	Max	$F1 + Opt$
1	Q_{num}	550*	1M	350	790	28%
2	S_{max}	800*	10K	2,200	10,000	18%
3	λ	900*	1M	3,800	16,000	19%
4	Z_{skew}	2.25*	100	14,000	60,000	18%
5	Ω	40*	1	5,600	16,000	26%

*Execution stopped with "out of memory" exception

cessing of overlapping data windows within a single aggregate (e.g., [12, 13, 6, 11, 24, 25, 18]) 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 order to minimize the cost of final aggregation, *TriOps* [7] uses intermediate function levels to pipeline partial aggregates to final-aggregate functions.

Under the multi-query optimization techniques, the general principle is to minimize the repeated processing of overlapping operations across multiple aggregate queries. This repetition occurs when queries exhibit an overlap in at least one of the following specifications: 1) predicate conditions, 2) group-by attributes, or 3) window settings.

Techniques leveraging overlaps in predicates and group-by attributes across *ACQs* are similar to classical multi-query optimization [16] that detects common subexpressions.

Techniques leveraging shared processing of overlapping windows across *ACQs* emerged with the paradigm shift for continuous queries. *Shared time slices* (*SLS*) [11] is one such a technique, which was also extended into *shared data shards* in order to share the processing of varying predicates, in addition to varying windows. Orthogonally, [15] extends classical subsumption-based multi-query optimization techniques towards sharing the processing of multiple *ACQs* with varying group-by attributes and similar windows.

Like *SLS*, *WeaveShare* [8] optimizes the shared processing of *ACQs* with varying windows by selectively partitioning them into execution trees resulting in a dramatic reduction in total processing costs. *WeaveShare* was also applied in distributed environments [17].

In [5], 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 [9, 10].

7. CONCLUSIONS

The main contribution of this paper is a novel closed formula, $F1$, for accelerating *Weavability* calculations required for determining the best execution plans for sharing partial aggregations of *ACQs*. Our approach replaces the counting of the edges within a *Bit Set* with mathematical computation. We proved theoretically that $F1$ significantly decreases the number of operations required for the execution plan generation while reducing the algorithm's space consumption to the bare minimum. We showed experimentally that the $F1$ approach achieves up to 60,000 times faster plan generation times compared to the current state of the art, and is able to achieve much better scalability in terms of the number of input *ACQs*, their diversity, and the input rate of the data stream. It should be noted that $F1$ can reduce the computation time of any optimization technique that requires scheduling partial aggregations within composite slides of multiple *ACQs*.

Acknowledgments We would like to thank Profs. K. Pruhs, J. Sorenson, and E. Bach for their help with our theoretical analysis and 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.

8. REFERENCES

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 2003.
[2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
[3] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
[4] C. Chung, S. Guirguis, and A. Kurdia. Competitive cost-savings in data stream management systems. In *COCOON*. 2014.
[5] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, 2010.
[6] 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.
[7] S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, 2012.
[8] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.
[9] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 2012.
[10] N. R. Katsipoulakis, C. Thoma, E. A. Gratta, A. Labrinidis, A. J. Lee, and P. K. Chrysanthis. Confidential elastic processing of data streams. In *SIGMOD*, 2015.
[11] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
[12] 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*, 2005.
[13] 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.
[14] 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.
[15] K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan. Memory-constrained aggregate computation over data streams. In *ICDE*, 2011.
[16] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
[17] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Processing of aggregate continuous queries in a distributed environment. In *BIRTE*. 2015.
[18] K. Tangwongsan, M. Hirzel, S. Schneider, K-L. Wu. General incremental sliding-window aggregation. *VLDB*, 2015.
[19] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.
[20] E. Weisstein. Binomial series. Wolfram MathWorld.
[21] E. Weisstein. Euclidean algorithm. Wolfram MathWorld.
[22] E. Weisstein. Harmonic number. Wolfram MathWorld.
[23] E. Weisstein. Prime number theorem. Wolfram MathWorld.
[24] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.
[25] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *VLDBJ*, 2010.