Multi-Query Optimization of Incrementally Evaluated Sliding-Window Aggregations

Anatoli U. Shein, *Member, IEEE* and Panos K. Chrysanthis, *Senior Member, IEEE*

Abstract—Online analytics, in most advanced scientific, business, and social media applications, rely heavily on the efficient execution of large numbers of Aggregate Continuous Queries (*ACQs*). *ACQs* continuously aggregate streaming data and periodically produce results such as *max* or *average* over a given window of the latest data. It has been shown that it is beneficial to use *Incremental Evaluation* (*IE*) for re-using calculations performed over parts of the *ACQ* window, and to share them in *multi-query* (*MQ*) environments among certain sets of *ACQs*. In this work, we re-examine how the principle of sharing is applied in *IE* techniques as well as in *MQ* optimizers. We provide an extensive taxonomy of *IE* techniques and a new approach of using the state-of-the-art *IE* techniques as part of *MQ* optimizers in a way that reduces the execution plan costs by up to 270,000x. We evaluate all of our solutions both theoretically and experimentally using both real and synthetic datasets.

Index Terms—Data streaming, sliding window, aggregate queries

1 Introduction

1.1 Motivation

Data stream processing has gained momentum in many applications that require quick responses based on incoming high velocity data flows. A representative example is a stock market application where multiple clients monitor the price fluctuations of the stocks. In this setting, a system needs to be able to efficiently answer analytical queries (e.g., average stock revenue, profit margin per stock, etc.) for different clients, each one with (potentially) different timing requirements. These requirements are typically associated with a *range* (or window) (**r**) and a *slide* (**s**), which can be either *tuple count* or *time*-based. A slide denotes the period at which an *ACQ* produces an output, and a range is the window over which the statistics are calculated.

Example 1. Consider a stock trading application monitoring average stock prices every 3 seconds for the past 5 seconds. Such an application submits a time-based *ACQ* with specifications of a 5 second range and 3 second slide.

Efficient data stream processing is also important in monitoring applications in the fields of health care, science, social media, and network control.

Data Stream Management Systems (*DSMS*) were proposed both in academia [1], [2], [3], [4], [5], [6], [7] and industry [8], [9], [10], [11], [12], [13], [14], [15] as the most suitable systems for handling such data flows on-the-fly and in real time. In a *DSMS*, clients register their analytical queries on incoming data streams. These queries continuously aggregate the incoming data, and as such they are called *Aggregate Continuous Queries* (*ACQs*).

1.2 Problem Statement

An *ACQ* requires the *DSMS* to keep state over time while performing aggregations. Normally, *DSMSs* only keep the window of the most recent data items, and when new data arrives, the window *slides* by discarding the data items that fall outside of the window specification and filling in the new data items. 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). It has been shown that in sliding-window stream processing it is beneficial to utilize *Incremental Evaluation* (*IE*), which operates by maintaining and reusing calculations performed over the unchanged parts of the window, rather than executing the *re-evaluation* of the entire window after each update [16], [17]. *IE* is also referred to as *Two-Ops* since it typically operates in two phases by (1) running partial aggregations on the data while accumulating it and (2) producing the answer by performing the final aggregation over the partial results [18], [19].

Recently, there were many advancements to the *Incremental Evaluation* (*IE*) of sliding-window queries. However, it has been shown that certain techniques are more beneficial than others in certain environments. For example *FlatFIT* [20] is optimized for increased throughput, while *DABA* [21] is optimized to maintain lower latency. Also, some techniques are only applicable for particular sets of aggregations, e.g., *Subtract-on-evict* [21] can only process invertible aggregations. Additionally, some *IE* techniques (such as *DABA*) focus solely on single-query scenarios where all computational resources are devoted to supporting one long-running, high accuracy *ACQ*, while others (such as *SlickDeque* [22]) also consider *multi-query* (*MQ*) settings that are common in multi-tenant *cloud* infrastructures, where multiple *ACQs* with a wide range of periodic features are executed on the same hardware and can share computation.

• *The authors are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260 USA. E-mail: {aus, panos}@cs.pitt.edu.*

Manuscript received 8 Dec. 2019; revised 28 Aug. 2020; accepted 29 Sept. 2020. Date of publication 9 Oct. 2020; date of current version 7 July 2022. (Corresponding author: Anatoli U. Shein.) Recommended for acceptance by C. Li. Digital Object Identifier no. 10.1109/TKDE.2020.3029770

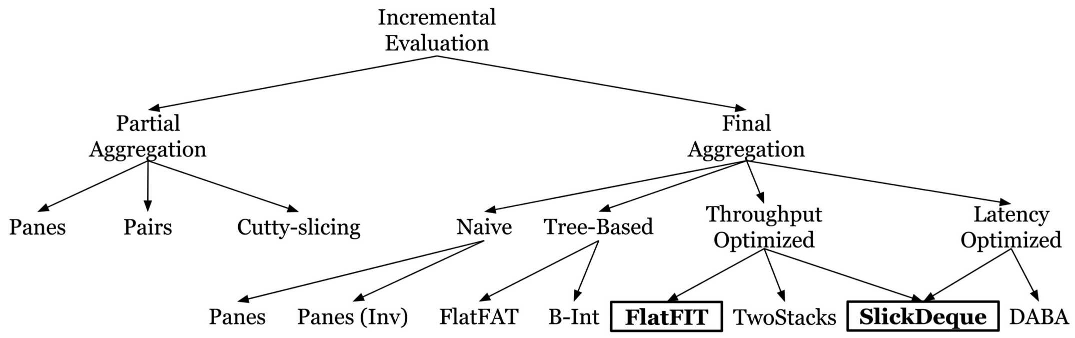

Fig. 1. Incremental Evaluation Taxonomy. Our contributions are marked with squares.

Also, most of the *IE* techniques that do consider *MQ* processing always force maximum sharing. However, it was shown that maximum sharing does not always offer maximum performance, and *selective sharing* may achieve better results by intelligently splitting the query load into multiple execution trees [23]. Unfortunately, the state-of-the-art *MQ* optimizers *WeaveShare* [23] and *TriWeave* [24] that support selective sharing only work with the outdated *Panes* [19] and *Pairs* [18] techniques (see Section 2) for *IE*. Thus, the opportunity arises to explore the suitability of new and more efficient *IE* techniques for use in combination with the *MQ* optimizers to achieve higher throughput and lower latency for processing large numbers of *ACQs*. Unfortunately, the task of combining new *IE* techniques with the *MQ* optimizers is hard since it requires an in-depth theoretical analysis of each compared *IE* technique towards estimating and quantifying its performance before the actual execution given any possible query workload. This challenge is addressed in this work.

1.3 Contributions

The contributions of this paper are as follows:

1) A taxonomy of all *IE* techniques available today, their breakdown in terms of applicability and goals, and our improvements to *IE* techniques and *MQ* optimizers (Sections 2 and 3).
2) A theoretical analysis of the available *IE* techniques that determines their average operational cost (Ω) per slide given any set of input *ACQs* and allows estimating their performance on average within 22 percent of the actual performance (Section 4).
3) Two new *MQ* optimizers based on *WeaveShare* and *TriWeave* that supports the new *IE* techniques using the performance estimates from the theoretical study above. The new optimizers reduce execution costs by up to 270,000x compared to the state-of-the-art ones (Section 5).

2 Incremental Evaluation Taxonomy

In order to provide a better context to our work, we developed a taxonomy of existing *Incremental Evaluation* (*IE*) techniques (illustrated in Fig. 1). The *IE* techniques can be broadly divided into *partial aggregation* and *final aggregation* categories. The *final aggregations* can be further distinguished into *Naive*, *Tree-based*, *Throughput Optimized*, and *Latency Optimized*.

Our taxonomy includes *IE* techniques that produce *exact* answers since it is crucial for many applications (e.g., financial, medical, etc.), and assumes in-order (or slightly out-of-order) *IE* processing. That is, our taxonomy does not consider approximate calculation methods, which were proposed to save time and space by sacrificing accuracy [25], [26], [27], [28], nor *IE* methods that handle out-of-order processing [29], [30]. Also our taxonomy does not include *IE* work in *Temporal Database Systems*, which stores the entire stream of tuples and allows aggregations over any continuous segments of the stream that are called *Historical Windows* [31], [32]. In contrast, we focus on techniques that process windows that end at or near the most recent results and are referred to as *Suffix Windows*. All the *IE* techniques in our taxonomy are compatible and can be used together with higher level general stream slicing and processing techniques [30], [33], [34].

Additionally, we only focus on distributive (simple) operations where the aggregation for the set S can be computed from two of the same aggregations of subsets $S1$ and $S2$, given subsets $S1$ and $S2$ were constructed by splitting S in two [35]. Calculating algebraic aggregations (which are composed of a number of distributive aggregations) follows trivially by maintaining several structures and/or applying extra calculations on final results, however we do not focus on them. The rest of the aggregations are holistic and are out of the scope for this work: they require specifically tailored algorithms which cannot be generalized.

2.1 Partial Aggregation

Partial aggregation can be thought of as the buffering of partial results until the query result needs to be returned by the final aggregation. Since partial aggregation allows buffering results that are later processed by a more expensive final aggregator, each buffered partial aggregate (or simply *partial*) is reused multiple times as part of different final aggregations, alleviating the use of CPU and memory resources. Clearly, it is beneficial to reduce the number of produced partials in order to minimize the amount of work done by the final aggregator. To this end the following partial aggregation techniques were proposed.

Panes [19] was proposed as the first partial aggregation technique for processing *ACQs* efficiently. The idea behind it is to partition the incoming datastream into "*panes*" (we refer to them as *partials*), and maintain just one aggregate

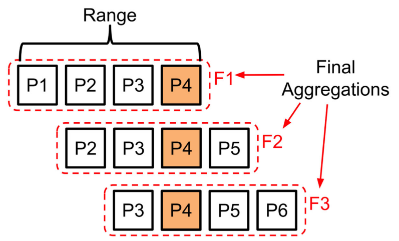

Fig. 2. Panes Technique (range=4 and slide=1).

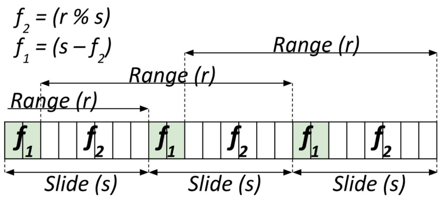

Fig. 3. Pairs Technique (range=14 and slide=8).

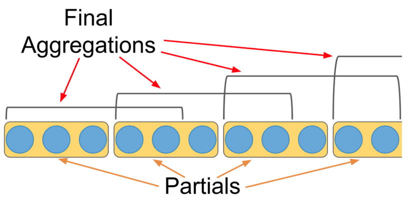

Fig. 4. Cutty-slicing Technique (range=5 and slide=3).

value for each partial. This way every incoming tuple affects the aggregate value for just the current partial, and when the whole aggregate is due to be reported, the answer is assembled by performing the final aggregation over all of the partials in the current window. Therefore, each new partial is reused multiple times for different final aggregations. For example, in Fig. 2 partial *P4* is used 3 times as part of the final aggregations *F1*, *F2*, and *F3*. The number of partials per window is $range/slide$ if the range is divisible by slide, otherwise it is $range/GCD(range, slide)$, where *GCD* is the Greatest Common Divisor.

Paired Windows (or simply *Pairs* [18]) was a technique introduced to reduce the number of partials in a window in cases where the range is not divisible by the slide. It works by splitting each slide into two fragments of different lengths as illustrated in Fig. 3, where fragment lengths f_1 and f_2, were calculated as follows: $f_2 = range\%slide$ and $f_1 = slide - f_2$. This way each window is composed of $2 \cdot \lfloor r/s \rfloor + 1$ partials, which significantly reduces the memory consumption and accelerates final aggregations.

Cutty-slicing was proposed as part of the *Cutty* optimizer [33]. The advantage of *Cutty-slicing* is that it starts each new partial only at positions that signify the beginning of new windows. This way the final aggregation can execute in the middle of the partial aggregation calculation by accessing the current value in the partial (Fig. 4). This reduces the number of partials per window to $\lceil r/s \rceil$ in cases where the range is not divisible by the slide at the cost of requiring a more complicated implementation.

Summary. Partial aggregation is beneficial in sliding window processing because it reduces the amount of more expensive final aggregations. In the setting where query

TABLE 1
Partial Aggregation Technique Comparison

Partial Aggregation Algorithm	# of partials per window when $r\%s = 0$	# of partials per window when $r\%s \neq 0$
Panes	r/s	$r/GCD(r,s)$
Pairs	r/s	$2 \cdot \lfloor r/s \rfloor + 1$
Cutty-slicing	r/s	$\lceil r/s \rceil$

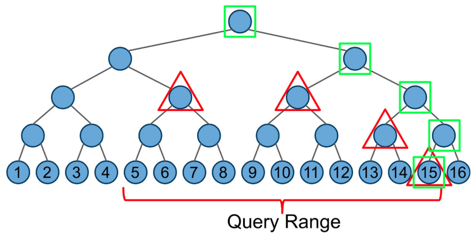

Fig. 5. FlatFAT technique.

ranges are divisible by their corresponding slides, all three of the above techniques perform the same, otherwise the *Cutty-slicing* technique achieves the best results. The comparison of the partial aggregation techniques is summarized in Table 1.

2.2 Final Aggregation

The goal of final aggregation is to produce *ACQ* results by assembling them from the partials.

Panes [19] (which we consider to be *Naive* in this work) works by simply iterating over the partials and constructing the answer. The example in Fig. 2 performs a final aggregation *F2* by iterating over partials *P2*, *P3*, *P4*, and *P5*. Naturally, such a solution quickly became outdated due to the increasing workloads that created bottlenecks in the final aggregator.

Panes (Inv) [19] (or Panes for Invertible/Differential Aggregate Queries) was proposed at the same time as *Panes* to efficiently process invertible aggregates. It works by maintaining a running aggregate (e.g., running *sum*), and invoking the inverse operation (e.g., *subtract*) on every expiring tuple. This algorithm (with minor differences) was also proposed as R-Int [36] and Subtract-on-Evict [21]. Despite being effective, Panes (Inv) is only applicable for invertible operations.

FlatFAT [37] (or Flat Fixed-sized Aggregator) is a final aggregation technique which stores tuples in a pre-allocated, pointer-less, tree-based data structure (Fig. 5). Originally, *FlatFAT* allowed only one tuple per leaf, however it was later extended [33] to perform partial aggregation by allowing it to store partial aggregates as tree leaves. Each internal node of the tree contains an aggregate of its two children. The root node has the result of the entire range allowed by the tree. In our experiments we compare our contributions to the improved version of *FlatFAT* [33].

New partials are inserted into the leaves of the binary tree left-to-right. The leaves form a circular array, meaning that after inserting a value to the rightmost leaf, the next insert goes into the leftmost one. Each insert triggers the update procedure, which is performed by walking the tree

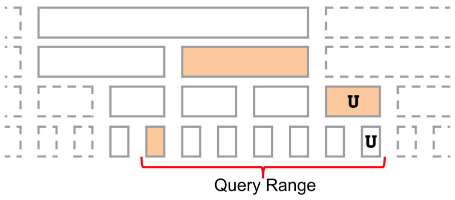

Fig. 6. B-Int technique.

bottom-up and updating all internal nodes with new aggregate values. An example of an update operation on leaf 15 is illustrated with green squares in Fig. 5. The look-up of the answer in *FlatFAT* is performed by returning the root node value if a query requires the result for the maximum window, or by aggregating a minimum set of internal nodes that covers the required range of leaf nodes. The example of answering a query with a range of 11 partials starting from leaf 15 is shown with red triangles in Fig. 5.

B-Int [36] (or Base Intervals) is another final aggregation technique that uses a multi-level data structure that consists of dyadic intervals of different lengths. On the bottom level the interval length is one partial, on the next level the interval length is two partials, on the third level the length is four partials, and so on until we reach the top level that just has one interval of the maximum supported range length. The whole data structure is organized in a circular fashion so that the rightmost interval on any level is followed by the leftmost interval from the same level (Fig. 6). The binary nature of this data structure makes it similar to *FlatFAT*, and like *FlatFAT*, when producing the final aggregate *B-Int* also determines the minimum number of intervals needed to represent the desired range and aggregates them. For example, in Fig. 6 *B-Int* aggregates all marked intervals to get the answer for the specified query range.

FlatFIT [20] (or Flat and Fast Index Traverser) was proposed with a goal of increasing the throughput of *ACQ* processing. *FlatFIT* achieves this acceleration by dynamically storing the intermediate results and their corresponding pointers indicating how far ahead *FlatFIT* can skip in its calculation. It uses two circular arrays *Pointers* and *Partials* interconnected with their indices and stack *Positions*. An example of update and look-up operations at position (or index) 5 is illustrated in Fig. 7. In this example we process a query with a range of 9 partials (which spans positions 8 to 5). To do that, *FlatFIT* follows the *Pointers* from position 8 to position 5, and pushes visited positions (8 and 1) on the *Positions* stack. Once position 5 is reached, all the *Partials* from the stored *Positions* are aggregated to return the final answer.

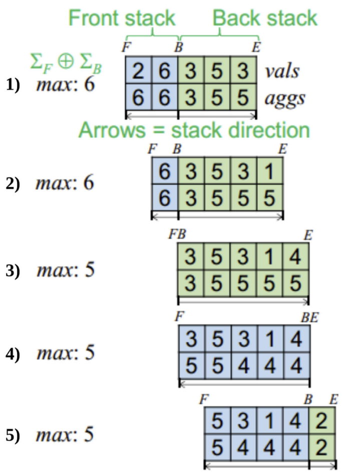

Fig. 8. TwoStacks technique.

While allowing high throughput *FlatFIT* has latency spikes due to the so called *window reset* event that occurs periodically every $n+1$ slides [20], rendering it a throughput optimized technique.

TwoStacks [21] was shown to also achieve high throughput by using an old trick from functional programming to implement a queue with two stacks, *F* (front) and *B* (back), where all insertions push a value *val* and an aggregation *agg* of everything below it onto *B*, and evictions pop from *F* as illustrated in Fig. 8. When *F* is empty (step 3 in Fig. 8), the algorithm flips *B* onto *F*, making it a calculation-heavy step that introduces latency spikes. To produce the final aggregation, the tops of both *F* and *B* stacks are aggregated, e.g., the *max* in step 2 is calculated as $max(6,1) = 6$.

DABA [21] (or De-Amortized Bankers Algorithm) was proposed as an alternative to *TwoStacks* that reduces the latency spikes while maintaining high throughput. The algorithm uses a principle of the Functional Okasaki Aggregator to de-amortize the *TwoStacks* algorithm. *DABA* uses two queues, *vals* and *aggs*, as shown in Fig. 9. These queues are implemented as chunked-array queues with six ordered pointers which make up the *F* and *B* stacks similarly to *TwoStacks*. However after each insertion and eviction event, a function *fixup* is called which re-balances pointers and fixes the consistency of the *aggs* queue. The details about how the *fixup* function performs the incremental reversal of *B* to *F* can be found in [21]. The implementations of *DABA* and several other techniques are also available on GitHub.

SlickDeque [22] unlike all the other *IE* techniques above does not process all *ACQs* *uniformly*. It handles aggregate operations differently based on their invertibility properties.

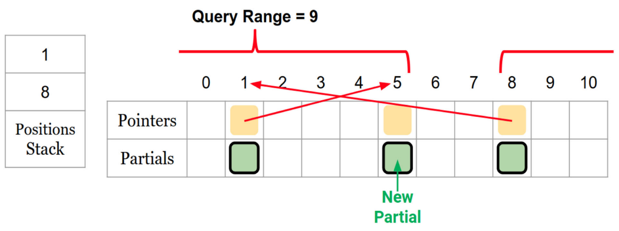

Fig. 7. FlatFIT technique.

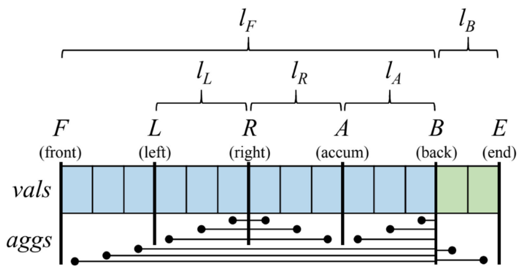

Fig. 9. DABA technique.

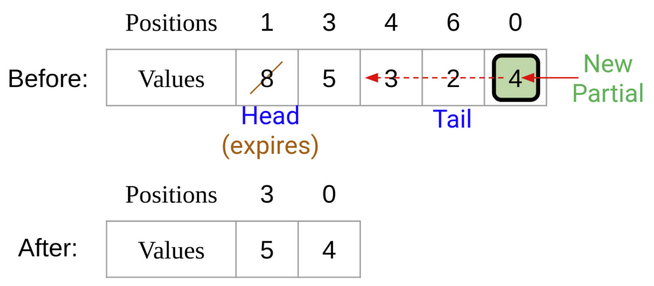

Fig. 10. SlickDeque (Non-Inv) technique.

TABLE 2
Algorithmic Complexities

Algorithm		Time			Space	
		Single Query		Max-Multi Query	Single Query	Max-Multi Query
		Amort	Worst			
Panes		n	n	n^2	n	n
FlatFAT		$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
B-Int		$log(n)$	$log(n)$	$n \cdot log(n)$	$2n$**	$2n$**
FlatFIT		3	n	$3n$	$2n$	up to $2.5n$
TwoStacks		3	n	—	$2n$	—
DABA		5	8	—	$2n$	—
Slick Deque	Inv	2	2	$2n$	n	$2n$
	Non-Inv	<2	n*	n	2 to $2n$*	2 to $2n$*

*the probability of these cases is negligible: 1 in $n!$.
**true only when n is a power of 2, otherwise $3n$.

The invertible operations are processed using *SlickDeque* (Inv), the modified *Panes* (Inv) approach, which allows multi-query processing by maintaining running aggregates for each unique range in a hashmap. Non-invertible *ACQs* are processed with a novel deque-based algorithm *SlickDeque* (Non-Inv), by requiring the following property: if operation $\oplus$ is *non-invertible*, then $x \oplus y = z$, where $z \in \{x, y\}$.

The intuition behind the *SlickDeque* (Non-Inv) algorithm can be seen in Fig. 10, which illustrates the moment when an update operation (insert partial 4 with sequential position 1) is performed on the deque structure that processes *max ACQs* with the slide of 1 partial and different ranges between 1 and 6 partials. First, since the new partial's position is 1 it means that a full iteration was just completed (1 through 6) and the previous partial with position 1 now expires and is removed from the head of the deque. Next, the new partial is added as the last element at the tail of the deque. However, since we are processing *max* and the value of the new partial is larger than the values 2 and 3, these partials are removed according to the algorithm, and the new partial is placed after the partial with value 5 and position 3. Next, we return the query answer (*max*). When the whole window is queried (to process *ACQ* with range 6) the *max* is located at the head of the deque (value 5 at this point). Otherwise, the answer is looked up by traversing the deque from head to tail and finding the partial with the correct (pre-calculated) sequential position.

Summary. The separation based on invertibility in *SlickDeque* leads to the best throughput and latency for both invertible and non-invertible operations in systems with heavy workloads out of all the above algorithms. In [22] we theoretically show the time and space complexity advantages of *SlickDeque* (summarized in Table 2) and experimentally validate them using a real workload. Specifically, *SlickDeque* achieves up to a 19 percent throughput improvement in a single query environment (Fig. 11) and up to a 345 percent improvement in a fully shared *MQ* environment (Fig. 12) while requiring up to 5 times less memory (Fig. 13) and maintaining 283 percent lower latency spikes on average (Fig. 14) over the next best approaches.

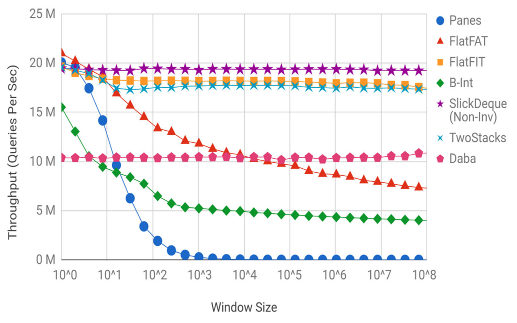

Fig. 11. Throughput in processed queries per second in single query environment (*max*).

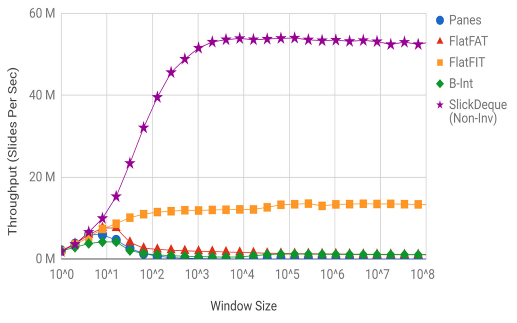

Fig. 12. Throughput in processed slides per second in multi-query environment (*max*).

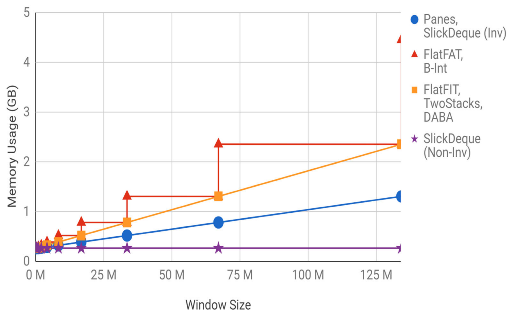

Fig. 13. Experimental memory usage in gigabyte increments.

With the introduction of the *SlickDeque* technique, the final aggregation for a single query can now be performed in constant time with no more than 2 operations per slide. Thus, we believe that sharing at the level of partial and final aggregation has reached its limit. In the rest of this work we focus on combining *IE* techniques with *MQ* optimization because it is the next logical step for further improving sliding-window aggregations.

3 MULTI-QUERY (*MQ*) OPTIMIZATION

The general objective of *MQ* optimization is to reduce (or eliminate) the repeated processing of overlapping operations

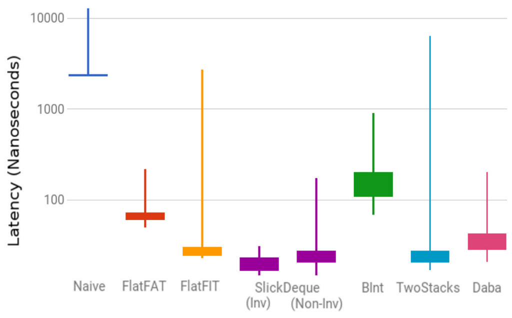

Fig. 14. Latency in nanoseconds per query answer. Solid boxes show values between 25th and 75th percentile, and lines show overall min and max values.

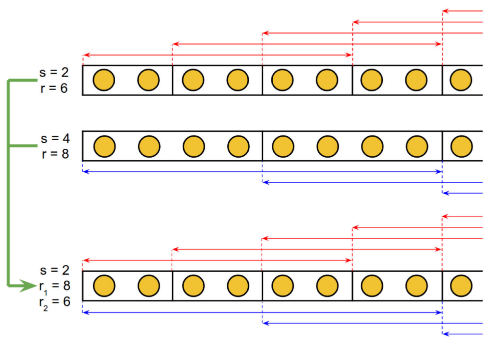

Fig. 15. Shared processing.

across multiple queries [38]. This repetition happens due to the processing of the same data items by different queries which exhibit an overlap in at least one of the following features: (1) predicate conditions, (2) group-by attributes, or (3) window specification. In this work we focus on optimizers targeting the window specification overlaps.

3.1 Shared Processing of ACQs

Since the *ACQs* are executed periodically (unlike one-shot, ad hoc queries), several processing schemes, as well as *ACQ* optimizers, take advantage of the shared processing of *ACQs* [18], [23], [29], [33], which reduces the long-term overall processing costs by sharing partial results. To show the benefits of sharing in such scenarios, consider the following example:

Example 2 (Fig. 15). Assume two *ACQs* monitor the *max* stock value over the same data stream. The first *ACQ* has a slide of 2 tuples and a range of 6 tuples, the second one has a slide of 4 tuples and a range of 8 tuples. That is, the first *ACQ* is computing partial aggregates every 2 tuples, and the second is computing the same partial aggregates every 4 tuples. Clearly the calculation producing partial aggregates only needs to be performed once every 2 tuples, and both *ACQs* can use these partial aggregates for their corresponding final aggregations. The first *ACQ* then only needs to run each final aggregation over the last three partials, and the second over the last 4.

Partial results sharing is applicable for all matching aggregate operations, (e.g., three different *ACQs* all calculating *max*), and for different but compatible aggregate operations (e.g., three different *ACQs* calculating *sum*, *count* and *average*, yet share results by treating *average* as $\frac{sum}{count}$).

To determine how many partial aggregates are needed after combining n *ACQs* into a shared execution plan, we first find the length of the new composite slide, which is the *Least Common Multiple* (*LCM*) of the slides of the combined *ACQs* (in Example 2 it is four). Each slide is then repeated $LCM/slide$ times to fit the length of the composite slide, and all slide multiples are marked within the composite slide as *edges* (in Example 2 the edges are at positions 2 and 4 of a composite slide). If slides consist of several fragments due to the partial aggregation, all fragments are also marked within the composite slide as edges. If two or more *ACQs* mark the same location, it means that location is a *common edge*. The more common edges that are present in the composite slide, the more partial aggregation sharing that can be performed.

Originally, the *Bit Set* technique [23] was used to determine how many partial aggregations (*edges*) are scheduled within the composite slide. This technique performs the counting of edges by traversing the entire composite slide and thus is very inefficient. Later we proposed an efficient mathematical solution to this problem, *Formula F1* [39] (described in Section 3.1.2).

3.1.1 Weavability

Out of all the *IE* techniques mentioned in Section 2, only *Panes*, *FlatFAT*, *B-Int*, *FlatFIT*, and *SlickDeque* are known to support *MQ* execution. These techniques share partial aggregates among all of the registered queries (i.e., all the queries are merged and processed as a single execution tree), thus achieving maximum sharing. However, it was shown that it is not always beneficial, and *selective sharing* achieves better performance by splitting the query load into multiple execution trees (that form an execution plan).

Intuitively, the overall cost of *ACQ* processing consists of partial aggregation and final aggregation costs. It is clear that combining *ACQs* together into execution trees helps to reduce the partial aggregation costs (since partials are shared across all *ACQs* in a tree and can be reused) while increasing the final aggregation costs (since partials themselves become smaller and require more frequent final aggregation processing). Thus, forcing maximum sharing by processing all *ACQs* in just one execution tree is likely to incur high final aggregation costs. To address the trade off outline above and quantify the plan costs the *Weavability* concept was proposed [23].

Weavability is a metric that measures the overall benefit of sharing intermediate results between any number of *ACQs*. If it is beneficial to share computations between these *ACQs*, then these *ACQs* are known to *weave* well together and are combined into the same shared execution tree by the *MQ* optimizer.

We say that two *ACQs weave* perfectly when their composite slide contains only *common edges* since this way when we process them together we minimize both partial aggregation costs (by executing just one tree) and final aggregation costs (by having partials of optimal length) at the same time.

The following formula can be used to calculate the cost (C) of the execution plan before and after combining *ACQs* into shared trees, and the difference between these costs tells us if the combination is beneficial:

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

where m is the number of the trees in the plan, λ is input rate in tuples per second, E_i is *Edge rate* of tree i, and Ω_i is the overlap factor of tree i. *Edge rate* is the number of partial aggregations performed per second, and the overlap factor is the total number of final-aggregation operations performed on each fragment. Note that both E_i and Ω_i are calculated by using ranges are slides of *ACQs* processed in tree i.

It is clear that $m\lambda$ represents partial aggregation costs since each input to each execution tree is processed by the partial aggregator. Similarly, $\sum_{i=1}^{m} E_i\Omega_i$ signifies final aggregation costs of the execution plan, where E_i determines how often the final aggregator will be invoked and Ω_i determines how much work it will need to do per each invocation.

Clearly, it is crucial to be able to generate high quality execution plans quickly. Unfortunately, this has been proven to be NP-hard [40], and currently only approximation algorithms can produce acceptable execution plans. Such approximation algorithms are utilized in the state of the art *MQ* optimizers *WeaveShare* [23] and *TriWeave* [24].

The *WeaveShare* [23] and *TriWeave* [24] *MQ* optimizers both utilize the concept of *Weavability* to produce execution plans for sets of input *ACQs*. The *TriWeave* optimizer was proposed as a part of a more general state-of-the-art *TriOps* [24] optimizer, which besides targeting window specifications (using *TriWeave*), also targets predicate conditions and group-by attributes.

Both *WeaveShare* and *TriWeave* optimizers selectively partition *ACQs* into multiple disjointed execution trees (i.e., groups), resulting in a dramatic reduction in the total query plan processing cost, and are theoretically guaranteed to approximate the optimal cost-savings to within a factor of four for practical variants of the problem [41]. Both *WeaveShare* and *TriWeave* start with a no-share plan, where each *ACQ* has its own execution tree. Then they iteratively consider all possible pairs of execution trees and combine those that reduce the total plan cost the most into a single tree, and produce final execution plans consisting of multiple disjointed execution trees when they cannot find another pair that would reduce the total plan cost further. The difference between *WeaveShare* and *TriWeave* is that the former assumes separate partial aggregation processing on each execution tree (causes larger partials), while the latter assumes combined partial aggregation processing using a large composite slide that passes ready partials to the execution trees (causes smaller partials). Because of this difference the execution plans produced by *WeaveShare* largely have higher partial aggregation costs and lower final aggregation costs compared to *TriWeave*. However, as we show in our experiments below, the overall costs of the execution plans produced by both optimizers are comparable (within 5 percent).

3.1.2 *F1: Accelerating the Optimization of* ACQs

Since the original *Weavability*-based *MQ* optimizers *WeaveShare* and *TriWeave* do not scale well with an increasing number of *ACQs*, we proposed a novel closed formula, *F1* [39], that accelerates *Weavability* calculations. *F1* efficiently calculates the number of unique *edges* in a composite slide allowing the *MQ* optimizers to achieve exceptional scalability in systems with heavy workloads:

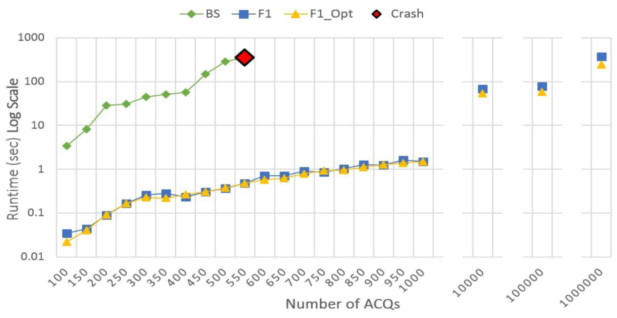

Fig. 16. Bit set and F1 scalability in terms of the number of ACQs using the WeaveShare optimizer.

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

where LCM_n is an LCM of all n processed slides s: $LCM(s_1, s_2, \ldots, s_n)$, and function $G_1(a,b)$ is a sum of the inversed *LCMs* of all possible groups of slides of size b from a set of size a. For example:

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

Note that calculating the number of unique *edges* in a composite slide is by far the most resource intensive part of *Weavability* calculations. This number was previously calculated using the *Bit Set* approach by iterating over the composite slide multiple times with different increments while identifying and *counting* the unique edges. *F1*, on the contrary, performs this task algebraically by *computing* them.

In general, $F1$ can reduce the computation time of any technique that combines partial aggregations within composite slides of multiple *ACQs*. We theoretically showed that $F1$ is superior in both time and space complexities to the previous *Bit Set* approach.

We showed that *Bit Set* approach has a time complexity of at least e^n, while $F1$ has a time complexity of at worst 2^n. Clearly, when n goes to infinity, it is increasingly beneficial to use $F1$ over *Bit Set* for producing the same execution plan given the same input. We experimentally showed that $F1$ executes up to 60,000 times faster and can handle 1,000,000 *ACQs* in a setting where the limit for the *Bit Set* technique is 550 (Fig. 16). In this experiment comparing the performance of Bit Set and F1, using both techniques, WeaveShare generated the exact same execution plan (consisting of the same number of trees and queries per tree) for each set of *ACQs*.

Summary. With the introduction of our *F1* formula for *Weavability* calculations, *MQ* optimizers are now able to achieve better scalability with increasing workloads. The modification of *MQ* optimizers in light of the new *IE* techniques are addressed in Section 4.

4 Utilizing New *IE* Techniques for *MQ* Optimization

It is intuitive that by combining new *IE* techniques and the *MQ* optimizers, significant benefits in *ACQ* processing can be achieved. This section presents how the existing MQ optimizers can be modified in order to support new efficient IE techniques. To accomplish this, the plan cost calculation process in *MQ* optimizers needs to be adjusted. The need for such adjustment can be extrapolated from Equation (1) for cost calculations discussed in Section 3.1.1.

Equation (1) uses the Ω parameter (which denotes the total number of final-aggregation operations performed per edge) calculated as range divided by slide per execution tree. As demonstrated in Section 2.2, such a number of final-aggregation operations is only applicable for the outdated *Panes* technique, and all other compared *IE* techniques perform fewer operations.

For the new *IE* techniques the Ω estimation is more complex due to the variability of operation numbers between different slides, and dependability on the input data, given that our *MQ* optimizers need to be able to estimate Ω for any number of *ACQs* with any periodical properties. Thus, a theoretical analysis (presented below) of all the *IE* techniques is necessary.

4.1 Estimating Ω

In order to evaluate how different *Incremental Evaluation* (*IE*) techniques perform when used in *multi-query* (*MQ*) optimizers, we need to calculate the number of final aggregation operations (Ω) that they perform on average per slide (i.e., after receiving each new partial) given Q unique *ACQs*. After that Ω is used in their corresponding cost formulas in *MQ* optimizers. The range and slide of each query q_i we denote as r_i and slide s_i respectively (i is a sequential number of a query). *IE* techniques must support *MQ* processing in order to be used in such optimizers, which rules out the *TwoStacks* and *DABA* techniques presented in Section 2.2. Our analysis of Ω for the rest of the techniques (*Panes*, *FlatFAT*, *FlatFIT*, and *SlickDeque*) follows below.

Panes. It is intuitive that Ω for this naive technique in single query environments can be calculated as range divided by slide (r/s) since the query range is assembled from r/s slides. Similarly, in *MQ* environments, since each added query increases Ω by its range divided by slide, we calculate it as follows:

$$\Omega = \sum_{i=1}^{Q} \frac{r_i}{s_i}. \tag{3}$$

FlatFAT. Given that this technique utilizes a binary tree for its calculations, in single query scenarios $\Omega = Q \cdot log_2(P_{max})$, where P_{max} is the total number of partials (or leaf nodes) in the tree. P_{max} is also the longest query range that can be processed by this structure. The Ω formula follows from the fact that the number of levels in a binary tree are $log_2(P_{max}) + 1$, and on each update *FlatFAT* updates the tree in a bottom-up fashion from the leaf to the root. The answer to the query with the longest range in this case could be simply taken from the root of the tree without additional operations. For each additional query with a unique range that consists of $P_i < P_{max}$ partials, the aggregate is composed from a minimum set of internal tree nodes that covers the number of partials (tree leaves) P_i. Thus, given that P_i can be calculated as r_i divided by the average partial length, each new query q_i increases Ω by $\lfloor log_2(P_i - 1)\rfloor + \frac{P_i - 1}{2^{\lceil log_2(P_i)\rceil}}$, resulting in the formula:

$$\Omega = \sum_{i=1}^{Q} (\lfloor log_2(P_i - 1)\rfloor + \frac{P_i - 1}{2^{\lceil log_2(P_i)\rceil}}). \tag{4}$$

Even though the actual number of partials processed is likely to be different for each slide, in the long run the cost per partial averages out, making this estimation valid. For quick approximate calculations Ω can be also estimated as $log_2(P_i - 1) + 1$ for each unique query.

FlatFIT. For this technique, we estimate Ω as $3 \cdot Q$, i.e., each unique *ACQ* requires about 3 operations per slide. We show in [20] that Ω is 3 operations per slide for a single query environment and $3 \cdot Q$ per slide in a *max-multi-query* environment (a *MQ* environment with the maximum number of queries covering all possible ranges from 1 to r_{max}). Thus, intuitively each added query should be adding 3 operations per slide to Ω. We confirmed this formula experimentally by testing a large number of various query sets. Even though there was slight variability in the results (due to the effect of periodic properties of *ACQs*), Ω always stayed close to $3 \cdot Q$ and never crossed $2 \cdot Q$ or $4 \cdot Q$. Given our intuition above, our closest estimation for *FlatFIT* appears to be:

$$\Omega = 3 \cdot Q, \tag{5}$$

which we use in our experiments with cost-based *MQ* optimizers.

SlickDeque (Inv). In single query environments this technique has $\Omega = 2$ because there are only two operations performed for each new partial: (1) the aggregation of the arriving partial with the running aggregate, and (2) the inversion operation of the expiring value (e.g., subtraction in the case of Sum). Similarly, in the case with multiple queries we get Ω by multiplying the number of running aggregates by 2. Unfortunately, the number of running aggregates does not always equal Q due to the different periodic properties of *ACQs*, an *ACQ* might assemble its final aggregate from different numbers of partials on different stages of execution, which means we need to keep running aggregates for all of these possibilities. However, if several queries need the same running aggregate (aggregating same number of partials) it is shared. Thus, in order to calculate the exact number of running aggregates required per query set we need to create a composite slide and iterate over it while counting all possible numbers of partials needed at every edge, and to get Ω we finally multiply the number of unique running aggregates by 2. Currently we do not know if there is a faster approach to determine this. Due to the complexity of this calculation in our experiments we use an approximation: first we divide each query range r_i by the average partial aggregate length to get P_i, and then take the count of all the unique P_i values and multiply by 2, resulting in the following formula:

$$\Omega = 2 \cdot \#\text{ of unique } P_i. \tag{6}$$

Given that in our experiments we generally have Q that is larger than the number of unique slides (which are generated by factoring a large number), the variance of P_i values is low, which makes our estimation valid (we also verified this experimentally), however this estimation might slightly vary in the other case.

SlickDeque (Non-Inv). Previously [22] we proved that Ω is bounded by 2 operations per slide (in single query environments), however in this work we worked out a more accurate estimation and accounted for *MQ* overhead in order to use this technique in *MQ* optimizers. *SlickDeque (Non-Inv)* performs exactly 2 operations per slide if we do not account for the following two cases: (1) expiration of partials at the head of the deque, and (2) deletion of the head node of the deque. When either of the two cases occur, 1 operation is performed for that slide instead of 2.

Case (1) happens when the partial stayed on the deque for the entire max query range worth of partials (P_{max}), which means that there was no input partials that could displace the expiring partial from the deque (e.g., if Max is calculated, there was not any input partial greater or equal to our expiring partial). The probability of that happening (given uniform input) is 1 to P_{max}, where P_{max} is the number of partials in the query with the longest range. Thus we subtract $1/P_{max}$ from Ω to account for the average number of times this happens in a long running process.

Case (2) happens when any input partial displaces the head node of the deque (e.g., if Max is calculated, a partial higher than all the nodes including the head node arrives). The probability of that happening is again $1/P_{max}$ per slide since that is the probability of the new partial displacing the most valuable partial from the latest P_{max} (e.g., the highest value if Max is calculated), thus we subtract another $1/P$ from Ω.

Now, in order to account for *MQ* cases we have to account for operations required by the algorithm to return query answers. In a single query environment this could be simply done by returning the value of the head node on the deque, however if we need to return answers to several queries with different ranges we traverse the deque. During the planning stage all queries are ordered descendingly by their ranges, which makes it possible during execution to get answers to all queries in just one full traversal of the deque. Each query requires at least one operation to compare its required P_i to the current iterator position within the deque. To account for that we add Q to our cost estimate Ω. Also, to account for the operations that need to be performed to traverse the deque (in the worst case) we add the number of operations equal to the average length of the deque during execution. Given the uniform input, as shown in [22], the average deque length equals the sum of the inversed factorials of sequential natural numbers from 1 to P_{max}, where P_{max} is again the maximum number of partials needed by any query to assemble its answer, and can be expressed as $\sum_{i=1}^{P} \frac{1}{i!}$. This follows from the fact that the probability of randomly picking x numbers ordered in a particular way (e.g., ascending) is 1 to $x!$. Thus, we estimate Ω for *SlickDeque (Non-Inv)* as:

$$\Omega = 2 - 2/P_{max} + Q + \sum_{i=1}^{P_{max}} \frac{1}{i!}. \tag{7}$$

Summary. We recap our theoretical findings in Table 3.

TABLE 3
Estimated Final Aggregation Costs

IE technique		Operations Per Edge (Ω)
Panes		$\sum_{i=1}^{Q} \frac{r_i}{s_i}$
FlatFAT		$\sum_{i=1}^{Q}(\lfloor log_2(P_i - 1)\rfloor + \frac{P_i - 1}{2^{\lceil log_2(P_i)\rceil}})$
FlatFIT		$3 \cdot Q$
Slick Deque	Inv.	$2 \cdot \#ofuniqueP_i$
	Non-I.	$2 - 2/P_{max} + Q + \sum_{i=1}^{P_{max}} \frac{1}{i!}$

5 Experimental Evaluation

In this section, we present the results of our experimental evaluation of using the new *IE* techniques in *MQ* optimizers by (1) generating execution plans for the *IE* techniques and comparing their estimated costs, and (2) actually executing several generated plans and comparing the practical performance.

5.1 Plan Generation Testbed

In this part of our evaluation we show the significance of *IE* technique selection on generated plan costs using our plan generating platform written in Java. Specifically, we implemented the *WeaveShare* and *TriWeave MQ* optimizers as described in [23] and [24] and augmented them with the support of different Ω calculations (estimation of final aggregations) for our compared *IE* techniques. Our workload is composed of a number of *ACQs* with different characteristics, which are generated synthetically in order to get a more detailed sensitivity analysis of the optimization performance by fine-tuning the generation parameters (described below). Moreover, it allows us to target all possible real-life scenarios and analyze them.

Plan generation experimental parameters:

[***IE technique***] Specifies which technique is used for *IE* processing. The available techniques are: *Panes*, *FlatFAT*, *FlatFIT*, *SlickDeque* (Inv and Non-Inv).

[$\boldsymbol{Q_{num}}$] Number of *ACQs*. We assume that all *ACQs* are installed on the same data stream, process distributive aggregations (which allow sharing partial aggregations among *ACQs*), and can be performed in just one operation.

[***Aggregations***] For simplicity we use *sum* for invertible, and *max* for non-invertible aggregations. As long as the actual aggregation functions are consistent with our assumptions, they do not have a significant effect on the final plan cost or processing performance.

[$\boldsymbol{S_{max}}$] Maximum slide length provides an upper bound on how large slides of our *ACQs* can be. The minimum slide allowed by the system is one. The slides are drawn from the set of factors of S_{max}.

[λ] The input rate describes how fast tuples arrive through the input stream in our system.

[$\boldsymbol{Z_{skew}}$] Zipf distribution skew 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.

[$\boldsymbol{O_{max}}$] Maximum overlap factor defines the upper bound for the overlap factor. The range of each *ACQ* is determined by drawing an ovelap factor from a uniform distribution between one and O_{max} and multiplying it by *ACQ*'s slide.

5.2 Plan Generation Results

To compare the sensitivity of the estimated plan costs produced by our new *IE* techniques to the parameters Q_{num}, S_{max}, O_{max}, λ, and Z_{skew}, we ran five experiments where we varied one of these parameters at a time while keeping the

TABLE 4
Experiment Parameters

Exp.#	Q_{num}	S_{max}	O_{max}	λ	Z_{skew}
1	*1-10K*	1K	10K	1	0
2	100	*10-100K*	10K	1	0
3	100	1K	*100-1M*	1	0
4	100	1K	10K	*0.01-100*	0
5	100	1K	10K	1	*(-1)-1*

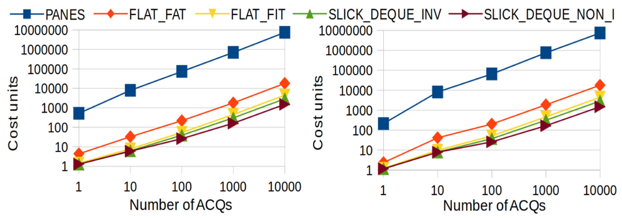

Fig. 17. Plan cost with increasing number of queries using WeaveShare (left) and TriWeave(right).

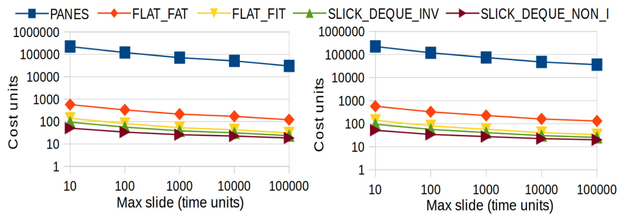

Fig. 18. Plan cost with increasing max slide using WeaveShare (left) and TriWeave(right).

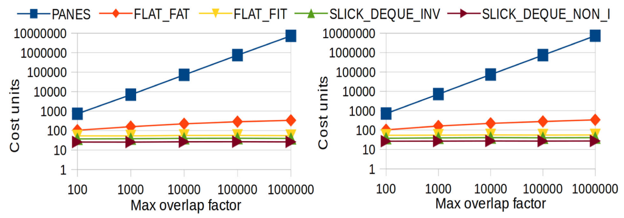

Fig. 19. Plan cost with increasing max overlap using WeaveShare (left) and TriWeave(right).

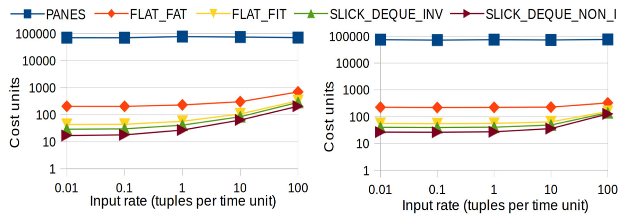

Fig. 20. Plan cost with increasing input rate using WeaveShare (left) and TriWeave(right).

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 five compared *IE* techniques. The experimental parameters are specified in the Table 4.

All results are taken as averages of running each experiment ten times. 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 Sensitivity

In this test we varied the Q_{num} from 1 to 10,000 (Fig. 17). Clearly increasing the Q_{num} also increases the amount of required calculations, causing higher costs for all of the generated plans (for both *WeaveShare*and *TriWeave* optimizers). The growth rates of all underlying *IE* techniques are similar to what we expected from the theoretical analysis of the time complexities of their underlying algorithms. Thus we see that using *SlickDeque* (Non-Inv) and *SlickDeque* (Inv) show the best results by outperforming the closest competing *IE* technique (FlatFIT) by up to 3x and the state-of-the-art *Panes* technique by up to 5,000x.

Exp 2: Max Slide Sensitivity

In this test we varied the S_{max} from 10 to 100,000 (Fig. 18). As opposed to to Exp 1, increasing the S_{max} decreases the amount of required calculations. This happens because with a higher max slide parameter, the generated *ACQs* have longer slides, which results in longer distances between the edges (where the final aggregations are performed). This way the workload for the final aggregator is reduced while keeping the same workload for the partial aggregator, resulting in a decreased total cost. Notice again that *SlickDeque* shows vastly superior performance by surpassing all other algorithms by up to 4,300x.

Exp 3: Overlap Factor Sensitivity

In this test we varied the O_{max} from 100 to 1,000,000 (Fig. 19). Similarly to Exp 1, increasing the O_{max} increases the amount of required calculations (in most cases). This follows from the fact that increasing O_{max} increases *ACQ* ranges, which require more partials to be assembled during each final aggregation. However, algorithms *FlatFIT* and *SlickDeque* (both Inv and Non-Inv) have constant complexity in terms of increasing window, thus their performance remains largely unaffected by the increasing ranges. As a result, we can see that the difference between the best performing *SlickDeque*technique and the currently used Panes technique grows much faster than in the first two experiments, and reaches 270,000x improvement.

Exp 4: Input Rate Sensitivity

In this test we varied λ from 0.01 to 100 (Fig. 20). Increasing λ increases the amount of required calculations because with higher input rates partial aggregators have to do more work aggregating the input tuples (which can be seen in Equation (1)). Notice that the performance of the *Panes* algorithm is not significantly affected by the increasing input rate because the cost of the *Panes* algorithm is largely dominated by the final aggregator cost, and the increase in partial aggregation cost is proportionally small. *SlickDeque* again outperforms other algorithms by up to 3,000x.

Exp 5: Slide Skew Sensitivity

In this experiment we varied the Z_{skew} from -1 to 1 (Fig. 21). It is similar to the max slide scalability experiment, because in both experiments we are gradually increasing the amount of *ACQs* with large slides and thus decreasing the amount of required calculations. The difference here is that when significantly skewing all slides drawn from the same set to one side (when Z_{skew} is close to -1 and 1), they start repeating, which lessens the affect on the costs (we can see flatter lines on the figures in these places). *SlickDeque* outperforms all the other *IE* techniques by up to 4,200x.

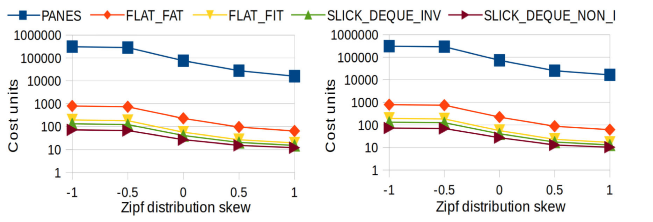

Fig. 21. Plan cost with increasing zipf using WeaveShare (left) and TriWeave(right).

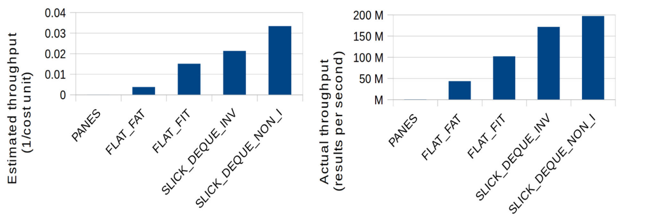

Fig. 22. WeaveShare: estimated throughput in 1/cost_unit per second (left), actual throughput in results per second (right).

Plan Generation Summary. Previously we showed that *SlickDeque* is best in stand-alone settings due to its superior algorithmic complexity [22]. The new results above confirm that *SlickDeque* still offers the same performance advantages when used in *MQ* optimizers. We also observed that it allows greater degree of weavability that results in fewer execution trees in an execution plan on average across all runs:

IE technique	Panes	FlatFAT	FlatFIT	SlickDeque
# of Execution Trees	16.0	15.7	10.3	4.8

This observed 2-3x reduction in the number of execution trees can explain the superior performance of *SlickDeque* in *MQ* optimizers. It also confirms our intuition that the more effective the *IE* technique is, the more benefit can be reaped by processing more *ACQs* together.

5.3 Practical Evaluation Testbed

In order to verify the correctness (and practical significance) of the plan cost estimations produced by our updated *MQ* optimizers in the first part of our evaluation, we executed a few selected plans on a real dataset using our execution platform written in C++ and examined their performance.

Setup. We generated one plan using *WeaveShare* and one plan using the *TriWeave* optimizer for each of the compared *IE* techniques using the query load parameters that correspond to the middle point of each figure from our plan generation experiments (Exp. 1-5):

Q_{num}	S_{max}	O_{max}	λ	Z_{skew}
100	1K	10K	1	0

Platform. For this evaluation we built an experimental platform in C++ (compiled with G++5.4). Specifically, we implemented a stand-alone stream aggregator platform and programmed the *Panes*, *FlatFAT*, *FlatFIT*, and *SlickDeque* (Inv and Non-inv) algorithms within the same codebase, sharing data structures and function calls to enable a fair comparison. Although all of the compared algorithms can be easily ported to any commercial general purpose stream processing system, we chose to go with a stand-alone platform to carry out our evaluation in an isolated environment in order to avoid any potential system interference and overheads. In the future we are planning to repeat our evaluation on a production system.

Dataset. We utilized the DEBS12 Grand Challenge Dataset [42] which contains events generated by sensors of large hi-tech manufacturing equipment. Each tuple in this dataset incorporates 3 energy readings and 51 values signifying various sensor states. The records were sampled at the rate of 100Hz, and the whole dataset includes ~33 million unique events.

Evaluation Metrics. Generally, a cost of a plan is estimated as the required computation power to process this plan. It is clear that there is a reverse relationship between the plan cost (measured in operations per second) and the throughput (measured as the number of actual query answers received per second). Thus, to perform a fair comparison we converted the plan cost to estimated throughput by inverting it (i.e., $1/Cost$).

5.4 Practical Evaluation Results

To measure the actual throughput, we ran each execution tree of a plan for 30 minutes at full speed while counting query results returned by the system, added them together to get the total number for the plan, and divided them by 1,800 to get the number per second. Even though the estimated and actual throughputs are measured in different units, they should correlate and be proportionally similar if our calculations are correct. These experiments were run on the same hardware as the plan generation experiments.

Exp 6: WeaveShare: estimated versus actual throughput

In this test we can see that visually our actual throughput correlates with the expected one, which gives us confidence that our updated *WeaveShare*optimizer performs cost estimation of all the *IE* techniques rather accurately (Fig. 22). The statistics say that if we normalize the scales of both estimated and actual throughputs by equating their largest readings, the average deviation between estimated and practical readings averages 31 percent, which is a good result given the dependency of the actual performance on a variety of system/environmental factors. Thus, we conclude that our estimations of the *IE* techniques are accurate enough to be used in the *MQ* optimizer *WeaveShare*.

Exp 7: TriWeave: estimated versus actual throughput

In this test we can again see that our actual throughputs are visually similar to the expected ones (Fig. 23). If we again normalize the scales of both estimated and actual throughputs, the deviation is on average only 14 percent,

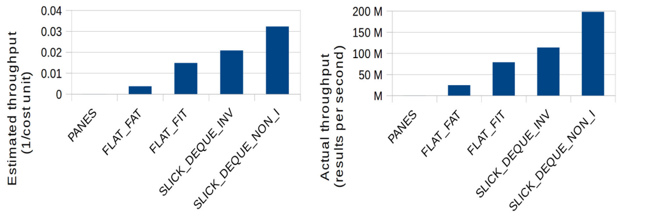

Fig. 23. TriWeave: estimated throughput in 1/cost_unit per second (left), actual throughput in results per second (right).

which is even better than in Exp. 6. Again we have confidence that our calculations have meaning and can be used in the *MQ* optimizer *TriWeave*.

Practical Evaluation Summary. The correlation that we see between the estimated and the actual performance numbers (with an average deviation of only 22 percent) gives us confidence that our new final aggregation cost calculations (Ω) are valid and can be utilized in cost-based *MQ* optimizers.

6 Conclusion

In this work we proposed a taxonomy of all *IE* techniques available today and their breakdown in terms of their applicability, complexity, and ability to work in *MQ* environments. The key contribution of this paper, however, is combining the recently developed *IE* techniques with the cost based *MQ* optimizers.

We addressed the challenging problem of theoretically analysing the *IE* techniques and estimating their performance dynamically based on any given workload. We used these findings in the state-of-art *MQ* optimizers for making tree collocation decisions, and experimentally showed that our estimations are within 22 percent of the actual numbers on average.

We also showed experimentally that *MQ* optimizers in combination with the *IE* technique *SlickDeque* produce up to 270,000x more efficient execution plans than in combination with other *IE* techniques, and we showed practically that our generated execution plans indeed achieve the expected (high) performance when executed on a real dataset.

References

[1] D. J. Abadi *et al*., "Aurora: A new model and architecture for data stream management," *Int. J. Very Large Data Bases*, vol. 12, pp. 120–139, 2003.

[2] D. J. Abadi *et al*., "The design of the borealis stream processing engine," in *Proc. 2nd Biennial Conf. Innovative Data Syst. Res.*, 2005, pp. 277–289.

[3] A. Arasu *et al*., "Stream: The stanford stream data manager," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, p. 665.

[4] M. A. Hammad *et al*., "Nile: A query processing engine for data streams," in *Proc. 20th Int. Conf. Data Eng.*, 2004, p. 851.

[5] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Algorithms and metrics for processing multiple heterogeneous continuous queries," *ACM Trans. Database Syst.*, vol. 33, 2008, Art. no. 5.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 379–390.

[7] S. Chandrasekaran *et al*., "TelegraphCQ: Continuous dataflow processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 668–668.

[8] A. Toshniwal *et al*., "Storm@twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.

[9] T. Akidau *et al*., "Millwheel: Fault-tolerant stream processing at internet scale," in *Proc. VLDB Endowment*, vol. 6, pp. 734–746, 2013.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," in *Proc. Int. Conf. Tech. Committee Data Eng.*, 2015, pp. 28–38.

[11] S. Kulkarni *et al*., "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, pp. 239–250.

[12] S. A. Noghabi *et al*., "Samza: Stateful scalable stream processing at linkedin," in *Proc. VLDB Endowment*, 2017.

[13] I. Brigadir, D. Greene, P. Cunningham, and G. Sheridan, "Real time event monitoring with trident," in *Proc. 1st Int. Workshop Real-World Challenges Data Stream Mining*, 2013.

[14] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proc. 4th USENIX Conf. Hot Topics Cloud Comput.*, 2012.

[15] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1123–1134.

[16] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid, "Incremental evaluation of sliding-window queries over data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 57–72, Jan. 2007.

[17] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 311–322.

[18] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 623–634.

[19] 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," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 39–44.

[20] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics," in *Proc. 29th Int. Conf. Sci. Statist. Database Manage.*, 2017, Art. no. 5.

[21] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 66–77.

[22] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "Slickdeque: High throughput and low latency incremental sliding-window aggregation," in *Proc. 21st Int. Conf. Extending Database Technol.*, 2018, pp. 397–408.

[23] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Optimized processing of multiple aggregate continuous queries," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, 2011, pp. 1515–1524.

[24] S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Three-level processing of multiple aggregate continuous queries," in *Proc. IEEE 28th Int. Conf. Data Engi.*, 2012, pp. 929–940.

[25] A. Bulut and A. K. Singh, "Swat: Hierarchical stream summarization in large networks," in *Proc. 19th Int. Conf. Data Eng.*, 2003, pp. 303–314.

[26] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, 2004, pp. 286–296.

[27] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, pp. 1794–1813, 2002.

[28] P. B. Gibbons and S. Tirthapura, "Distributed streams algorithms for sliding windows," in *Proc. 14th Annu. ACM Symp. Parallel Algorithms Architectures*, 2002, pp. 63–72.

[29] K. Tangwongsan, M. Hirzel, and S. Schneider, "Optimal and general out-of-order sliding-window aggregation," *Proc. VLDB Endowment*, vol. 12, pp. 1167–1180, 2019.

[30] J. Traub *et al*., "Scotty: Efficient window aggregation for out-of-order stream processing," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1300–1303.

[31] B. Moon, I. F. V. López, and V. Immanuel, "Scalable algorithms for large temporal aggregation," in *Proc. 16th Int. Conf. Data Eng.*, 2000, pp. 145–154.

[32] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," in *Proc. 17th Int. Conf. Data Eng.*, 2001, pp. 51–60.

[33] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 1201–1210.

[34] J. Traub *et al*., "Efficient window aggregation with general stream slicing," in *Proc. 22nd Int. Conf. Extending Database Technol.*, 2019, pp. 97–108.

[35] J. Gray *et al*., "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining Knowl. Discovery*, vol. 1, pp. 29–53, 1997.

[36] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 336–347.

[37] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proc. VLDB Endowment*, vol. 8, pp. 702–713, 2015.

[38] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[39] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "F1: Accelerating the optimization of aggregate continuous queries," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, 2015, pp. 1151–1160.

[40] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava, "Multiple aggregations over data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 299–310.

[41] C. Chung, S. Guirguis, and A. Kurdia, "Competitive cost-savings in data stream management systems," in *Proc. Int. Comput. Combinatorics Conf.*, 2014, pp. 129–140.

[42] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic, "The debs 2012 grand challenge," in *Proc. 6th ACM Int. Conf. Distrib. Event-Based Syst.*, 2012, pp. 393–398.

Anatoli U. Shein (Member, IEEE) received the BS degree from Duquesne University, in 2011, and the MS and PhD degrees from the University of Pittsburgh, in 2018 and 2019, respectively. He was a member of the Advanced Data Management Technologies Laboratory, University of Pittsburgh and is currently a software engineer with Vertica Pittsburgh, where he focuses on integration with Data Lakes. His research interests include the areas of data management, databases, and data streams.

Panos K. Chrysanthis (Senior member, IEEE) received the BS degree from the University of Athens, Greece, and the MS and PhD degrees from the University of Massachusetts at Amherst. He is a professor of computer science and the founding director of the Advanced Data Management Technologies Laboratory, University of Pittsburgh. He is also an adjunct professor with the Carnegie Mellon University and at the University of Cyprus. His research interests include the areas of data management (Big Data, Databases, Data Streams & Sensor networks), distributed & mobile computing, operating systems and real-time systems. He received the US NSF CAREER Award and he is an ACM distinguished scientist.