FlatFIT: Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics

Anatoli U. Shein
Department of Computer Science
University of Pittsburgh
aus@cs.pitt.edu

Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh
panos@cs.pitt.edu

Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

ABSTRACT

Data stream processing is becoming essential in most current advanced scientific or business applications as data production rates are increasing. Different companies compete to efficiently ingest high velocity data and apply some form of computation in order to make better business decisions. In order to successfully compete in this environment, companies are focusing on the most recent data within a count or time-based window by continuously executing aggregate queries on it. Incremental sliding-window computation is commonly used to avoid the performance implications of re-evaluating the aggregate value of the window from scratch on every update. The state-of-the-art *FlatFAT* technique executes *ACQs* with high efficiency, but it does not scale well with the increasing workloads. In this paper we propose a novel algorithm, *FlatFIT*, that accelerates such calculations by intelligently maintaining index structures, leading to higher reuse of intermediate calculations and thus exceptional scalability in systems with heavy workloads. Our theoretical analysis shows that *FlatFIT* is superior in both time and space complexities compared to *FlatFAT*, while maintaining the same query generality. Given a window of size n, *FlatFIT* achieves *constant* algorithmic complexity compared to $O(log(n))$ complexity of *FlatFAT*. We experimentally show that *FlatFIT* achieves up to a *17x* throughput improvement over *FlatFAT* for the same input workload while using less memory.

CCS CONCEPTS

•Information systems → Data streams; Online analytical processing; •Theory of computation → Streaming, sublinear and near linear time algorithms;

KEYWORDS

FlatFIT, Aggregate Continuous Query, Sliding-Window Processing

ACM Reference format:
Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2017. FlatFIT: Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3085504.3085509

1 INTRODUCTION

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 (i.e., average stock revenue, profit margin per stock, etc.) for different clients, each one with (possibly) different relaxation levels in terms of accuracy. Apart from financial applications, efficient data stream processing is important in fields such as health care, science, social media, and network control.

Data Stream Management Systems (*DSMS*) [1–3, 20, 25] have been proposed as 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 streaming data, and as such 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* [14]), and can be either count or time-based. A slide denotes the period at which an *ACQ* updates its answer; a range is the window for which the statistics are calculated. For example, if a stock monitoring application has a slide of 3 seconds and a range of 5 seconds, it means that the application needs an updated result every 3 seconds, and the result should be derived from data accumulated over the past 5 seconds.

An *ACQ* requires the *DSMS* to keep state over time while performing aggregations. Normally, *DSMSs* only keep the window of the most recent data, and produce the query answers by running different aggregation queries over it. When new data arrives, the window slides by discarding the data that falls out of the window specification and filling in the new data. This allows the aggregate query to execute over the updated window and reflect recent changes. Since it has been shown that in sliding-window stream processing it is beneficial to reuse unchanged parts of the underlying window, the idea of incremental evaluation is becoming more and more attractive compared to the window re-evaluation after each update [9, 18]. Often, it is useful to run partial aggregations on the data while accumulating it, which could be thought of as buffering, and then produce the answer by performing the final aggregation over the partial results [16, 17]. 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 Efficient handling of aggregate operations that are *non-invertible* and *non-commutative* proved to be essential in calculation heavy domains such as finance and science. Examples

SSDBM '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3085504.3085509

include Max, Min, Concatenate, First N, Last N, CountDistinct, CollectDistinct, ArgMax, and ArgMin.

This paper focuses on such non-invertible or non-commutative operations that are heavily used in practical *ACQs*. We consider both **single query** environments where each *ACQ* executes in isolation, for example for privacy reasons, and **multi-query** environments, where a large number of *ACQs* with different periodic properties (accuracies) are operating on the same data stream, calculating similar aggregate operations. An example of a multi-query environment is a multi-tenant *DSMS* deployed to a *Cloud* Infrastructure, where multiple *ACQs* with a wide range of different periodic features are executed on the same hardware.

The current state-of-the-art solution for efficiently processing these kinds of workloads is the Reactive Aggregator framework implemented using the *Flat Fixed-sized Aggregator* (also known as *FlatFAT*) [24]. *FlatFAT* is able to achieve high throughput by utilizing a pre-allocated memory circular tree-based data structure, however it does not scale well with heavy workloads. Recently, a new system, *Cutty* [7], was proposed that utilizes *FlatFAT* in a multi-query environment and contributes a novel slicing technique (referred to as *Cutty-slicing* in the rest of the paper) for partitioning the incoming tuples. However it does not improve the main query processing technique which is *FlatFAT*.

To address the aforementioned shortcomings, in this paper we propose a novel solution named *Flat and Fast Index Traverser*, or simply *FlatFIT*, which accelerates the processing of *ACQs* by significantly speeding up the final aggregation operation of incremental sliding-window evaluation techniques. *FlatFIT* achieves this acceleration by maintaining intermediate aggregates in intelligent indexing structures that reduce the number of partials used in performing a final aggregation and allows a greater level of reuse of previously calculated results. We show both theoretically and experimentally that our approach allows better scalability in terms of window size, and it becomes advantageous to utilize *FlatFIT* over *FlatFAT* starting with windows of a size as small as eight tuples (or partials in cases when partial aggregation techniques are used).

Contributions We make the following contributions:

- We propose a novel solution for processing *ACQs*, *FlatFIT*, which supports both non-invertible and non-commutative aggregate operations and is applicable for both single query and multi-query environments.
- We theoretically evaluate the proposed *FlatFIT* approach and mathematically show that it achieves a time complexity of $\boldsymbol{O(1)}$ (compared to $\boldsymbol{log(n)}$ complexity of the state-of-the-art *FlatFAT* approach) and a space complexity of $\boldsymbol{2n}$ (compared to $2^{\lceil log(n) \rceil + 1}$ complexity of *FlatFAT*). To our knowledge, there are no prior algorithms that can achieve the same asymptotic time and space complexities without losing the query generality in terms of supported aggregate operations.
- We experimentally evaluate the *FlatFIT* approach based on a real dataset and show that it significantly outperforms the state-of-the-art *FlatFAT* technique in most applicable scenarios by increasing the *ACQ* throughput by up to 17 times while reducing memory consumption by up to 1.9 times, making *FlatFIT* significantly more effective in processing *ACQs* in an on-line *DSMS*.

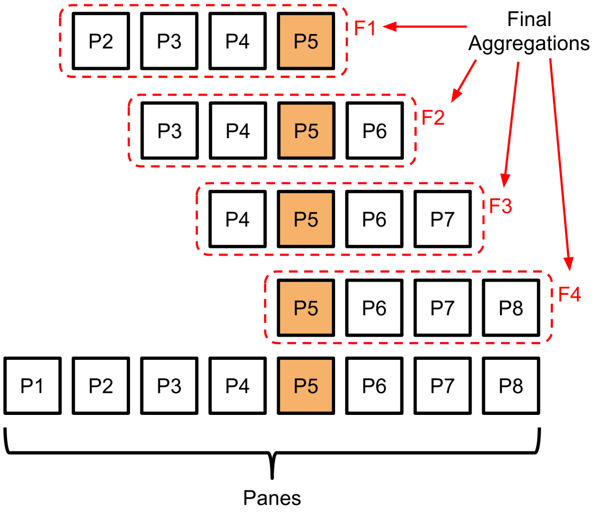

Figure 1: Panes Technique

Roadmap In the next section, we provide related work which constitutes the background of our work. We introduce our new technique, *FlatFIT* for the final aggregation calculations in Section 3. The complexity analysis of *FlatFIT* and compared algorithms is presented in Section 4. We discuss the evaluation platform and the experiments in Section 5 and conclude in Section 6.

2 BACKGROUND & RELATED WORK

In this section we briefly review the underlying concepts of our work, which are the incremental sliding-window computation techniques. These could be broadly divided into *partial aggregation* and *final aggregation*. We also review other related work.

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. It is clearly only beneficial when the answer look-up is not scheduled to happen after every single update according to the query semantics (i.e., *ACQ* allows some buffering if its slide is greater than 1 tuple). Since partial aggregation allows some buffering before the result needs to be processed by a more expensive final aggregator and each buffered partial can be reused multiple times as part of final aggregations, the use of the CPU and memory resources can be alleviated. Because of this, the slide plays a crucial role in determining the amount of partial aggregations that can be done. Currently, the following techniques are commonly used for partial aggregations: *Panes*, *Pairs*, and *Cutty-slicing*.

Panes [17] 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 value for each partial. This way every incoming tuple will affect 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 the partials in the current window. Therefore, each new partial

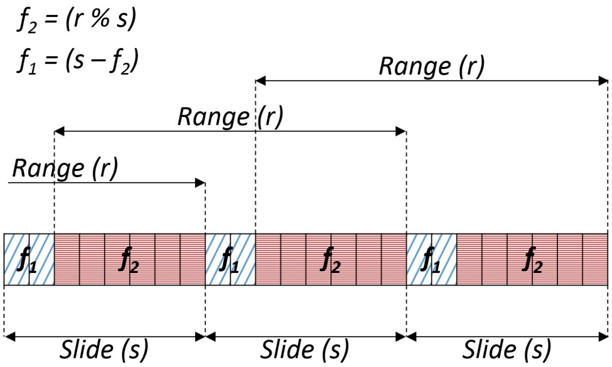

Figure 2: Paired Window Technique

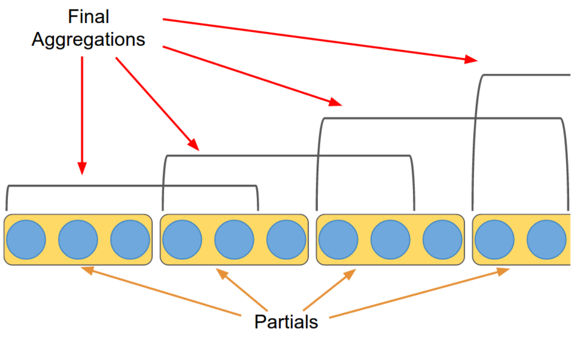

Figure 3: Cutty-slicing Technique

will be reused multiple times for different final aggregations. For example, in Figure 1 partial $P5$ is used 4 times as part of the final aggregations $F1$, $F2$, $F3$, and $F4$. 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 Window technique or simply *Pairs* [16] was introduced to reduce the number of partials in a window in cases when the range is not divisible by the slide. This technique makes the memory consumption twice as small and accelerates the final aggregations by reducing the number of partials by a factor of 2. As illustrated in Figure 2, two fragment lengths are used, f_1 and f_2, where $f_1 = range\%slide$ and $f_2 = slide - f_2$. The final aggregations are computed interchangeably each time after fragment f_2 is computed.

Cutty-slicing was proposed as part of the *Cutty* optimizer [7]. 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 aggregations can execute in the middle of partial aggregation calculations. The final aggregation just uses the current value in the partial, and after it is done, the partial resumes its calculation (Figure 3). This technique reduces the number of partials per window down to $\lfloor range/slide \rfloor + 1$, which is twice as small as in the *Pairs* approach. However it comes at a cost: instead of having offline execution plans specifically tailored to the query, *Cutty-slicing* has to send additional punctuations over the data stream to the execution module in order to indicate the beginnings of the new partials, which reduces the effective bandwidth of the stream and can significantly slow down the system, especially if the workload includes a large number of queries with small windows. To avoid such complications, in this paper we utilize the *Pairs* approach for the partial aggregations.

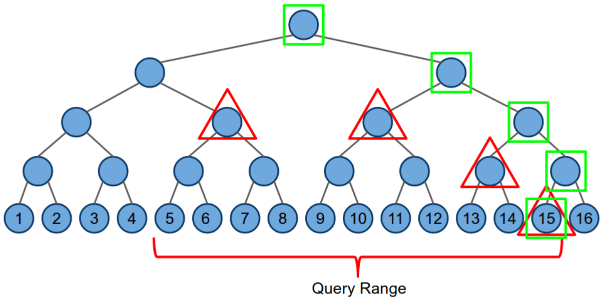

Figure 4: FlatFAT Technique

2.2 Final Aggregation

The goal of final aggregation is to produce the result of a query by utilizing the partials. Initially it was done by simply iterating over them and constructing the answer [16, 17]. For example the *Panes* technique in Figure 1 performs a final aggregation $F1$ by iterating over partials $P2$, $P3$, $P4$, and $P5$. Naturally, the naive solution quickly became outdated due to the increasing workloads that created bottlenecks in the final aggregator. In order to improve this, several final aggregation techniques have been proposed [5, 19, 23, 24, 26]. Out of the aforementioned techniques only *FlatFAT* and *B-Int* satisfy our complexity and query generality requirements.

FlatFAT is the state-of-the-art approach for final aggregations, which stores tuples in a pre-allocated pointer-less tree-based data structure (Figure 4). Originally, *FlatFAT* allowed only one tuple per leaf, effectively preventing itself from doing partial aggregations (since effectively the slide is one tuple for any *ACQ*). However a new general sliding-window processing solution, *Cutty*, extended *FlatFAT* to allow processing multiple queries within the same window 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 maximum range allowed by the tree. In this work we use the improved version of the *FlatFAT* algorithm, which allows partial aggregates to be stored in the tree leaves, and we simply refer to it as *FlatFAT* for clarity.

The algorithm works by sequentially inserting new partials into the leaves of the binary tree left-to-right. The leaves by themselves form a circular array, meaning that after inserting a value to the rightmost leaf, the next insert will go into the leftmost one. Each insert triggers the update procedure, which is performed by walking the tree bottom-up and updating all internal nodes with new aggregate values. The update finishes when the root node is updated. An example of an update operation on leaf 15 is illustrated with green squares in Figure 4.

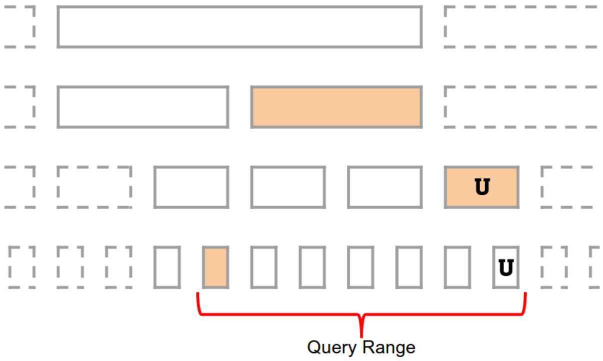

Figure 5: B-Int Technique

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 in leaf nodes. The example of answering a query with range of 11 partials starting from leaf 15 is shown with red triangles in Figure 4.

B-Int (or Base Intervals) was proposed in [5] as another final aggregation technique. It uses a multi-level data structure that consists of dyadic intervals of different lengths. On the first level, the intervals are of a length of 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 (Figure 5). Notice that the binary nature of this data structure makes it similar to *FlatFAT*, and similarly to *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 Figure 5 *B-Int* aggregates all intervals marked with color in order to get the answer for the specified query range. Yet, the algorithms for updates and look-ups are slightly different. During insertions, unlike *FlatFAT*, *B-Int* only updates the intervals that end with the inserted value instead of updating the entire structure bottom up until reaching the top layer. This, however, slows down look-ups since more intervals are needed to be aggregated to get the result. Due to the algorithms' similarities, *B-Int* and *FlatFAT* share the same time complexities $O(logn)$, and space complexities $2^{\lceil log(n) \rceil + 1}$, however *B-Int* was shown to be slower than *FlatFAT* by a constant factor in [24], and we confirm these findings in this work as well.

L-Int and **R-Int** [5] were proposed together with *B-Int*, however they are not applicable to our work because they do not satisfy our complexity and query generality requirements. *R-Int* requires queries to be both commutative and invertible, and *L-Int* is only applicable for cases with large numbers of historic look-ups compared to the number of updates, where historic look-ups return answers for ranges within a window that do not include the latest data. Conversely, the *L-Int* algorithm becomes impractical since its complexity devolves to $O(n)$ in such a setting.

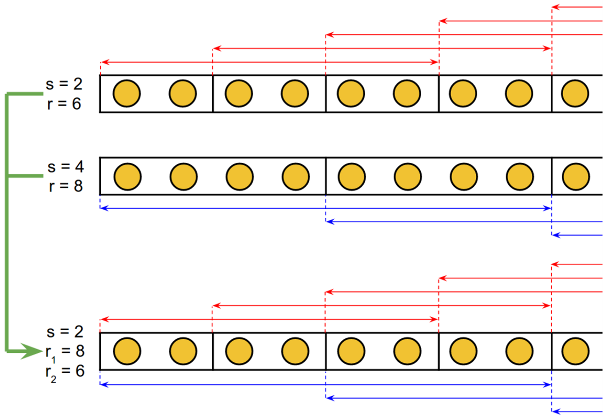

Figure 6: Shared Processing

2.3 Shared Processing of ACQs

Since the *ACQs* are executed periodically (unlike one-shot queries), the opportunity to reduce the long-term overall processing costs by sharing partial results arises. Several processing schemes, as well as *ACQ* optimizers, take advantage of the shared processing of *ACQs* [7, 13, 16]. To show the benefits of sharing partial aggregations in such scenarios, consider the following example:

Example 1 (Fig. 6) Assume two *ACQs* that monitor *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* will then run each final aggregation over the last three partial aggregates, and the second *ACQ* will run each final aggregation over the last 4 partial aggregates.

Partial results sharing is applicable for all matching aggregate operations, such as *max*, *count*, *sum*, *average*, etc., and for different but compatible aggregate operations, for example *sum*, *count* and *average* can share results by treating *average* as *sum/count*.

To determine how many partial aggregations are needed after combining *n* *ACQs* into a shared execution plan, we need to first find the length of the new combined (composite) slide, which is the *Least Common Multiple* (*LCM*) of the slides of the combined *ACQs* (in Example 1 it is four). Each slide is then repeated $LCM/slide$ times to fit the length of the new composite slide, and all slide multiples are marked within the composite slide as *edges*. If slides consist of several fragments due to the used 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 are present in the composite slide, the more partial aggregation sharing can be performed.

In this work we combine all compatible *ACQs* into one shared plan in order to achieve maximum sharing, which, in a general case, provides the most computational resource savings. Although, in specific cases it was shown that it is not always beneficial to aim for maximum sharing [12, 13, 21].

2.4 Other Related Work

Work similar to sliding-window aggregation existed in *Temporal Database Systems* long before *DSMSs* came around. Such systems store the entire stream of tuples and allow aggregations over any continuous segments of the stream which are called *Historical Windows*. Conversely, *DSMSs* generally only support *Suffix Windows*, which end at or near the most recent results. In the context of Temporal Databases, Moon et al. [19] utilized red-black trees for aggregations and Yang et al. [26] used SB-trees, which incorporate features from both segment-trees and B-trees. Due to the tree-based natures of these algorithms their update complexities are $O(log(s))$, where s is the size of the entire stream history over which they build their structures. Additionally, they do not allow non-invertible aggregations, which significantly restricts their applicability.

Several approximate calculation approaches were proposed to save time and space by giving up accuracy [4, 6, 8, 10]. Our approach focuses solely on computing *exact* answers since it is crucial for many applications (i.e., financial, medical, etc.).

In order to improve the partial aggregation sharing in a multi-query scenario, several heuristic-based plan optimizers have been proposed for single node systems (WeaveShare [13], TriOps [12], $F1$ [21]) as well as for distributed environments [22]. We consider all these techniques complimentary to ours since they can be applied directly on top of our multi-query $FlatFIT$ aggregator.

3 FLATFIT

In this section we describe our new algorithm, $FlatFIT$, that significantly speeds up the final aggregation calculations in a sliding-window environment.

3.1 Algebraic Properties and Assumptions

One of the important metrics that allows the evaluation of the difficulty of incremental evaluation of a particular query is the algebraic properties of the underlying aggregate operation. Based on classification from [11], we divide all aggregate operations into three broad categories: *distributive*, *algebraic*, and *holistic*.

- **Distributive** aggregation means that the aggregation for the set S can be computed from two of the same aggregations of subsets $S1$ and $S2$, where subsets $S1$ and $S2$ were constructed by splitting S in two. For example, if we have a set of 10 numbers and the sum of the first 7 is 20, and the sum of the 3 remaining is 15, then we can get the sum of all 10 numbers by adding 20 and 15. Therefore, sum is a distributive aggregation.
- **Algebraic** aggregation means that the aggregation can be computed from a number of distributive aggregations, i.e., *average* is an algebraic aggregation because we can calculate it from two distributive aggregations: sum and count. The list of common distributive aggregations includes count, sum, product, max, min, sum of squares, etc. By combining these distributive aggregations we can calculate some algebraic aggregations commonly used in statistics such as: average (count and sum), standard deviation (sum of squares, sum, and count), geometric mean (product and count), and range (max and min).
- **Holistic** aggregations are aggregation that are neither distributive nor algebraic, i.e., median, top-K, quantile. Holistic aggregates are out of the scope for this work since they require specifically tailored algorithms which cannot be generalized.

In this paper we will focus just on optimizing the distributive aggregations, since calculating the algebraic aggregations follows trivially. Distributive aggregations can be further classified by their mathematical properties: *associativity*, *invertibility*, and *commutativity*. Below we provide brief definitions of these properties.

- An operation $\oplus$ is *associative* if $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ is true for all x, y, z.
- An operation $\oplus$ is *invertible* if there exists an operation $\ominus$ such that $(x \oplus y) \ominus y = x$ for all x, y, and $\ominus$ is feasibly inexpensive.
- An operation $\oplus$ is *commutative* if $x \oplus y = y \oplus x$ is true for all x, y.

Query Operation Assumptions The state-of-the-art approach, $FlatFAT$, as well as our proposed approach, $FlatFIT$, both support non-invertible and non-commutative operations, however they require the operation to be associative. In general, all operations that can be executed on a window of values are associative. The common non-associative operations such as subtraction ($x - y - z$), division ($x/y/z$), exponentiation (x^{y^z}), vector cross product ($\vec{x} \times \vec{y} \times \vec{z}$), and some binary operations such as $NAND$ and NOR, are generally impractical when executed on sets of values larger than two.

Window Structure Assumptions In *non-FIFO* window structures, the events of insertion and expiration are not synchronized, which can cause window overflow situations. Such cases arise when there are not enough expiring tuples (or partial aggregates) to make room in the window for the insertions. Both the $FlatFAT$ and the $FlatFIT$ approaches are able to handle such cases by performing dynamic resize operations, however in this paper we are focusing on the *FIFO* window environment since it is the most common approach to processing sliding-window aggregations in practice, and we treat *non-FIFO* windows as a special case.

Arrival Order Assumptions Similarly, both algorithms allow updates on multiple partial aggregates already stored within the window. However in this paper we focus on the classic streaming scenario when all new partial aggregates are processed by the final aggregator one-by-one as they become available. In such settings the arriving tuples have to be *in-order* or slightly *out-of-order*. As long as the *out-of-order* tuples are within the same partial aggregation, the final result will not be affected. If, however, the ordering of tuples is further degraded, inconsistencies in the final result may arise. The mechanism that our system uses to cope with such extreme situations is outside of the scope of this paper.

3.2 The FlatFIT Algorithm

In this subsection we provide the algorithm and implementation details for our approach followed by two clarifying examples. We target single query and multi-query environments, though single query can be considered a special case of multi-query processing.

Our *FlatFIT* algorithm works by intelligently reusing calculations performed on sets of partial aggregates within a window,

Algorithm 1 FlatFIT Pseudocode

```
1:  Input: A set of aggregate continuous queries Q, aggregate operation ⊕, the initial value for ⊕ initVal, and partial aggregation technique PAT
2:  Output: Continuous answers to queries in Q according to their specifications.
3:
4:                        Phase 1 (Preparation)
5:  sharedPlan = BuildSharedPlan(Q, PAT)
6:  wSize = sharedPlan.wSize
7:  Partials = new array[wSize]
8:  Pointers = new array[wSize]
9:  Positions = new stack()
10: for i=0 to wSize do
11:     Partials[i] = initVal
12:     Pointers[i] = i + 1
13: end for
14: Pointers[wSize - 1] = 0
15: currInd = 0
16: prevInd = wSize - 1
17:
18:                        Phase 2 (Execution)
19: while results are expected do
20:     length = sharedPlan.getNextPartialsLength()
21:     newPartial = PartialAggregator.aggregate(length, PAT)
22:     Partials[prevInd] = newPartial
23:     Pointers[prevInd] = currInd
24:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
25:     for each query q in queriesToAnswer do
26:         startInd = currInd - q.range
27:         if startInd < 0 then
28:             startInd += wSize
29:         end if
30:         do
31:             Positions.push(startInd)
32:             startInd = Pointers[startInd]
33:         while startInd != currInd
34:         end do while
35:         answer = Partials[Positions.pop()]
36:         while Positions.size() > 1 do
37:             tempInd = Positions.pop()
38:             answer = answer ⊕ Partials[tempInd]
39:             Partials[tempInd] = answer
40:             Pointers[tempInd] = currInd
41:         end while
42:         tempInd = Positions.pop()
43:         answer = answer ⊕ Partials[tempInd]
44:         send (answer)
45:     end for
46:     prevInd = currInd
47:     currInd++
48:     if currInd == windowSize then
49:         currInd = 0
50:     end if
51: end while
```

which are stored in a novel index structure. In this way, the FlatFIT algorithm avoids costly and unnecessary (re)computations and enables a higher reuse of the intermediate results than previous methods. Pseudocode for the *FlatFIT* algorithm is depicted in Algorithm 1 and consists of the *Preparation* and *Execution* phases.

The Data Structures in the core of the *FlatFIT* algorithm include two circular arrays (*Pointers* and *Partials*) and a stack (*Positions*). The two circular arrays are interconnected with their indices (as shown in Figure 7), and the stack is used to store the indices that are currently processed. The *Pointers* and *Partials* arrays can be thought of as a single weighted jump table that allows *FlatFIT* to skip to the position stored in the *Pointers* array while adding the corresponding value from the *Partials* array to the running aggregate value. This way some calculations that have been already accounted for can be skipped.

The Preparation Phase given a set of queries Q and one of the partial aggregation techniques discussed in Section 2.1 (i.e., *Pairs*) as an input, starts by building a shared execution plan by executing the *BuildSharedPlan* function (line 5). The *sharedPlan* is constructed as discussed in Section 2.1, and it includes a full list of partials (or edges) augmented with their lengths and lists of queries that need to be evaluated at each partial. The *BuildSharedPlan* function identifies the query with the longest range in terms of the number of partials, and saves this range as the member *wSize* of the produced *sharedPlan*. *wSize* signifies the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *FlatFIT* initializes the data structures (lines 7-14). The two circular arrays are both initialized to a length equal to *wSize*. The *Positions* stack is initialized empty and can expand up to *wSize* − however normally it is much less (refer to Section 3.3). The *Partials* array is initially filled with the initial value *initVal* for the query operation ⊕ supplied as input. For example, *initVal* is 0 for the *SUM* operation or $-\infty$ for the *MAX* operation. Each value in the *Pointers* array is initialized to point to the next consecutive value in it (i.e., $Pointers[2]$ is 3, and $Pointers[wSize - 1]$ is 0, since it is a circular array).

The *currInd* variable signifies the current position within the two arrays (line 15). It starts at 0 initially and increases to $wSize - 1$ during execution, after which it wraps back to 0. The arriving partial aggregates will be inserted into the *Partials* array always at the index previous to the *currInd*, referred to as *prevInd* (line 16).

The Execution Phase is implemented as a loop that continuously returns all the query results while they are expected. At the beginning of the loop (lines 20-24), *FlatFIT* gets the next partial’s length from the *sharedPlan*, and supplies it to our *Partial Aggregator* which uses the provided *PAT* technique to produce the *newPartial* value. The *newParial* is then inserted into the *Partials* array at *prevInd*, and $Pointers[prevInd]$ is updated to point to the *currInd*. Now, the answers to all queries scheduled at this position need to be produced.

After receiving the *queriesToAnswer* from the *sharedPlan* (which is a subset of Q), *FlatFIT* loops over these queries in order to answer them. The loop starts by identifying the start index *startInd* for each query q (lines 26-29) within the two arrays from which it will start aggregating values. *startInd* is identified by rewinding *currInd* back by q’s range length.

Once the *startInd* of *q* has been determined, our algorithm traverses the *Pointers* array while pushing all visited indices onto the *Positions* stack in a do-while loop until it reaches the *currInd* again (lines 30-34). Then, in order to construct the final aggregation *FlatFIT* needs to access the *Partials* array at all these indices, and at the same time update the values in the *Partials* array to be reused in the future.

Towards this (lines 35-44), *FlatFIT* first initializes the *answer* variable to the value found in the *Partials* array at the index popped from the top of the *Positions* stack. It then continues by popping all the indices except for the last one from the *Position* stack in a loop and saving them as a *tempInd*. The values found at the *tempInd* indices in the *Partials* array are aggregated with the *answer* variable using the aggregate operation ⊕ supplied as an input. Each time a new partial is aggregated, *FlatFIT* also writes the current value of the *answer* into the *Partials* array at *tempInd*, and copies the *currInd* into the *Pointers* array also at *tempInd*. This technique allows *FlatFIT* to later skip from *tempInd* to *currInd* by doing just one aggregate operation. The last index popped from the *Positions* stack is also used to retrieve the corresponding partial from the *Partials* array and is aggregated to the *answer*, however it does not need to update the two arrays because it will be overwritten in the next iteration of the execution phase with the new partial.

Observations. Notice that the more queries with different ranges that are registered on the datastream, the more result reusing is performed by *FlatFIT*. In cases where the number of queries registered on the datastream is small, large parts of the *Pointers* and *Partitions* arrays might be visited and updated by *FlatFIT* on certain slides (not more frequently than once per *wSize*), which enables fast calculations on the rest of the window.

The least amount of calculation reuse for the *FlatFIT* algorithm happens in a single query environment, since once per $wSize + 1$ all indices are visited and pushed onto the *Positions* stack, which then causes an update on almost the entire window. In this paper, we refer to this event as *wReset*. *wReset* also happens as the first iteration of the execution phase in any environment regardless of how many queries are registered on the datastream. Even though a single query environment turns out to require the most computation for *FlatFIT*, it still significantly outperforms all competitors including the state-of-the-art *FlatFAT* technique. This stands because despite *wReset* being a heavy calculation part, it only happens once per $wSize + 1$ and it enables *FlatFIT* to reuse calculated partials efficiently during the rest of the execution.

The following Examples 2 and 3 (illustrated in Fig. 7) should clarify the above algorithm. In order to make the explanation more intuitive we execute the two queries $Q1$ and $Q2$ on the same incoming datastream using two algorithms: *Naive* and *FlatFIT*, and we illustrate each step of their calculations side-by-side.

Example 2 (Single query environment). Assume we have just one query $Q1$ which is seeking the *MAX* value over the range of 5 tuples with a slide of 1 tuple. The slide size is set to one tuple in this example for simplicity, which means that there is no partial aggregation and the answer to *MAX* needs to be calculated after every new tuple arrival. A shared execution plan is not needed in this example since we only have one query, which makes our window size (*wSize*) equal to the range of $Q1$ (5 tuples).

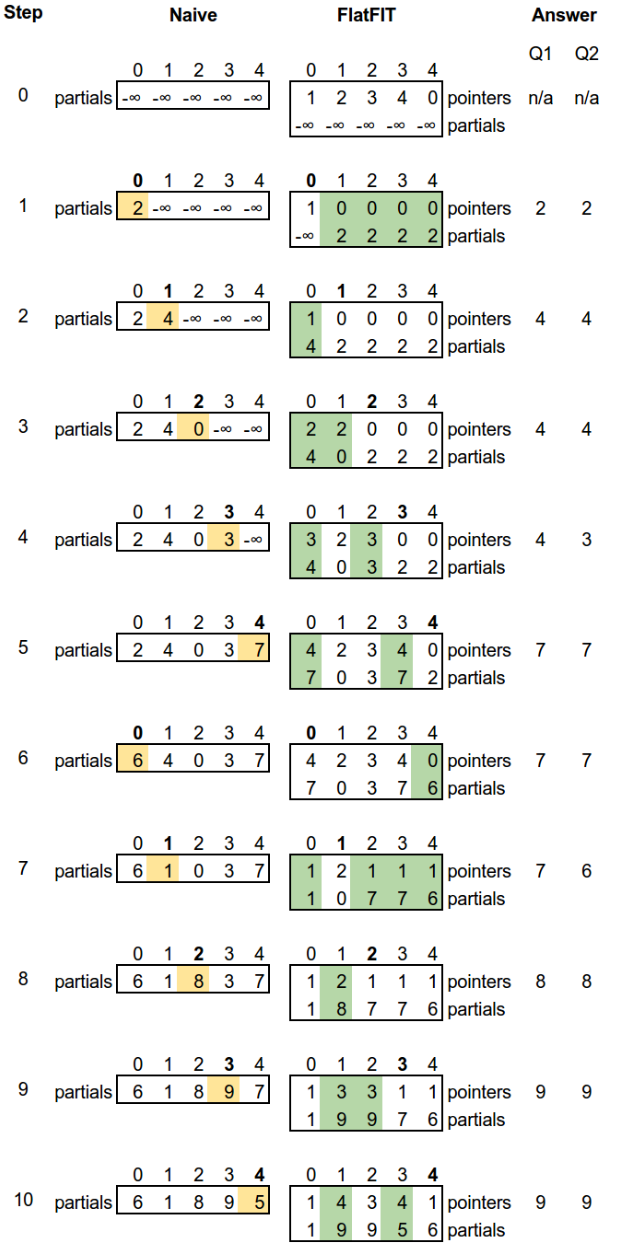

Figure 7: Example of Naive and FlatFIT algorithms working in a Single Query Environment (processing just Q1) and in a Multi-Query Environment (processing both Q1 and Q2)

In both the *Naive* and *FlatFIT* representations we mark the positions that have been modified by the algorithms in each step. The *Positions* stack involved in the *FlatFIT* calculation is not illustrated here, however its contents in each step are clear since we know that all indices that are modified in that step were pushed onto the *Positions* stack and then popped back off. The current index (*currInd*) at each step is bolded in Figure 7 for convenience. The tuples enter the system in the order: 2, 4, 0, 3, 7, 6, 1, 8, 9, 5.

After the initialization in Step 0, in Step 1 the first tuple, 2, arrives. The *Naive* algorithm stores the new tuple at the current index in its own *Partials* array, and it executes a full iteration over the entire array in order to find the *MAX* value, which in this case is 2.

FlatFIT writes the first tuple, 2, to the *Partials* array at the previous-to-the-current index, which in this case is 4 (we refer to this index as *prevInd*). Now the algorithms have to make a full circle over the *Pointers* array because in a single query environment, the start index (*startInd*) for the query is always equal to the *currInd*. By the nature of the *FlatFIT* operation discussed above, Step 1 always triggers the *wReset* event (the update of the whole window except for the current index) because the *Pointers* at each index are pointing to the next index after the initialization, and *FlatFIT* is unable to skip any positions while producing the result. This way, all indices are pushed onto the *Positions* stack and subsequently popped to construct the answer from the partials at those indices, while also updating the arrays for future use. Thus, all indices (except the *currInd*) are pointing now to index 0 and their corresponding values in the *Partials* array are set to 2.

In Step 2, the *Naive* algorithm places the new partial, 4, into the current index and iterates again over the whole window comparing every value in order to get the *MAX* value (which now is 4). Our *FlatFIT* algorithm is able to provide the answer to $Q1$ here with just one *MAX* comparison. From the start index, 1, it skips to index 0 (since *Pointers*[1] is 0) and then again to index 1 since *Pointers*[0] is 1. The answer is then computed by taking the *MAX* of *Partials*[1] and *Partials*[0], which is 4, and it is then stored in *Partials*[0].

In Step 3 *FlatFIT* updates index 1 with the new tuple, 0, and it is able to make a full circle from the *currInd*, 2, back to itself by visiting intermediate indices 0 and 1, after which just index 0 was updated for future use.

In Steps 4, and 5 (and later 9) *FlatFIT* is able to get the answers in just two *MAX* comparisons similarly to Step 3, and in Steps 6 and 8 it takes just one comparison similarly to Step 2, while *Naive* did 4 comparisons at each and every step. Step 7 forced *FlatFIT* to execute 4 comparisons similarly to *Naive* because the *wReset* event happens at this step.

In a single query environment the *wReset* happens on the first inserted partial and then repeats periodically every $wSize + 1$ slides. Since the period is greater than *wSize* by one, the start position of the *wReset* operation keeps shifting right by one every cycle. ■

Notice that this small example highlights the benefit of using *FlatFIT* over *Naive* by showing that *Naive* had to execute 40 MAX comparisons total to process $Q1$, while *FlatFIT* executed just 21.

Example 3 (Multi-query environment). In this example we illustrate how *FlatFIT* works in a multi-query environment by augmenting Example 2 with one more query, $Q2$. The new query, $Q2$, is also seeking the *MAX* value and has a slide of 1 tuple, however its range is 2 tuples. Thus, *Naive* and *FlatFIT* will need to answer both queries at every step. Since the range of $Q1$ is 5, which is greater than the range of $Q2$, and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a *wSize* of 5 tuples.

The *Naive* algorithm in this case does a full loop over the entire array in order to answer $Q1$ each time, and then iterates over the most recent two partials to produce the answer for $Q2$, and this process is repeated at every step.

Conversely, *FlatFIT*, after iterating over the whole structure in Step 1 to produce the answer for query $Q1$, is able to generate the answer for $Q2$ with 0 comparisons, just by calculating the start index, *startInd*, for $Q2$ (which is 3) and reading the answer from the *Partials* array at this index (since the *Pointers* array at this index points us directly back at the *currInd*). Similar behavior for calculating the answer for $Q2$ in 0 comparisons can be also found in Steps 3, 7, and 9.

In Step 2, our *FlatFIT* algorithm calculates the answer for $Q1$ just by doing one comparison (explained in Example 2), and produces the answer for $Q2$ by executing also just one *MAX* comparison (of *Partials*[4] and *Partials*[0]). Similarly to this step, *FlatFIT* calculated the answers for query, $Q2$, in just one comparison also in Steps 4, 5, 6, 8, and 10. ■

Notice that even for query, $Q2$, with range as small as 2 tuples, *FlatFIT* needed just 6 comparisons for the entire example, while *Naive* had to perform 10. It is intuitive that with increasing query numbers and their ranges, *FlatFIT* allows much better scalability. Later in this paper this intuition is backed up by both theoretical analysis (Section 4) and experimental evaluation (Section 5).

3.3 Optimization

In order to reduce memory consumption by the *Positions* stack in a single query environment we made the following observation: the stack fills up to $wSize - 1$ only during the *wReset* event, otherwise it can hold up to 2 values at most. In fact, the usage of the *Positions* stack repeats with period $wSize + 1$ and it always contains $wSize - 1$ entries at the first step of each cycle, one entry at the second and the last entries of the cycle, and two entries in the rest of the $wSize - 2$ steps. This means that the amount of memory consumed by the *Positions* stack can be reduced from $wSize - 1$ to 2 by implementing the *wReset* operation manually without using the stack.

In our optimized *wReset* function, we initialize the *answer* variable to the initial value, *initVal*, for the query operation ⊕, and iterate over both the *Partials* and *Pointers* arrays of *FlatFIT* backwards from *prevInd* until *currInd* is reached. At each iteration the value from the *Partials* array is aggregated into the *answer* variable, and the current value of the *answer* variable is written back to the *Partials* array. The *Pointers* array is updated to point to the *currInd* at each iteration. After the traversal is finished, the value from the *answer* variable is returned, and both arrays of *FlatFIT* are updated and ready to continue executing the main algorithm.

This manual *wReset* function is triggered periodically every $wSize + 1$ slides in a single query environment, and triggered just once at the beginning of the execution phase of multi-query environments. The full implications of this optimization on an algorithm's space complexity can be found in Section 4.2.

4 COMPLEXITY ANALYSIS

In this section, we calculate the time and space complexities of *Naive*, *B-Int*, *FlatFAT*, and *FlatFIT*, summarized in Table 1.

The theoretical time complexities of the above algorithms are illustrated in Figures 8 & 10, theoretical throughputs in Figures 9 & 11, and theoretical memory consumptions in Figure 12.

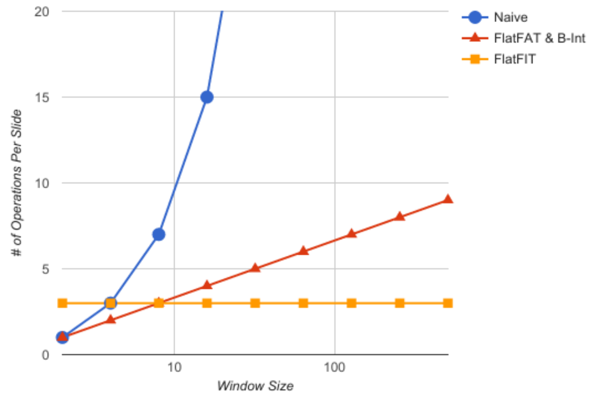

Figure 8: Theoretical operations per slide in a single query environment

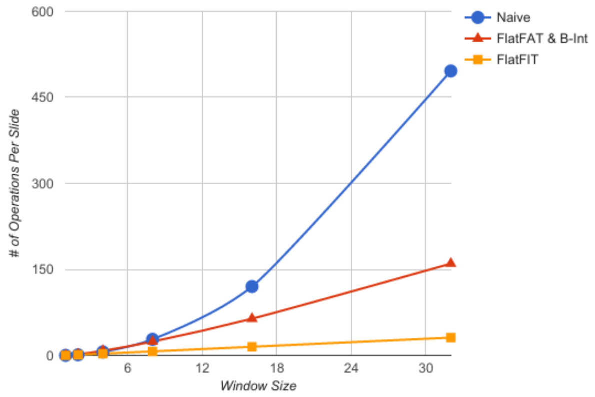

Figure 10: Theoretical operations per slide in a max-multi-query environment

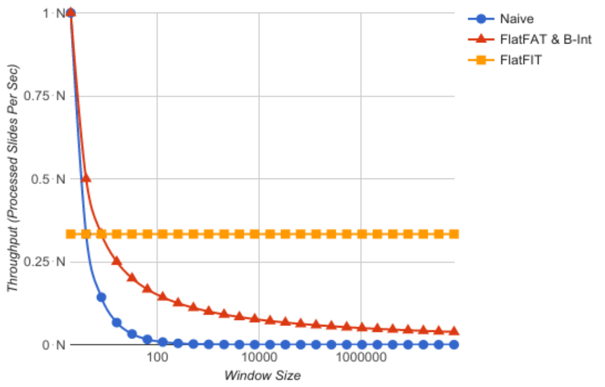

Figure 9: Theoretical throughput in a single query environment running N operations per second

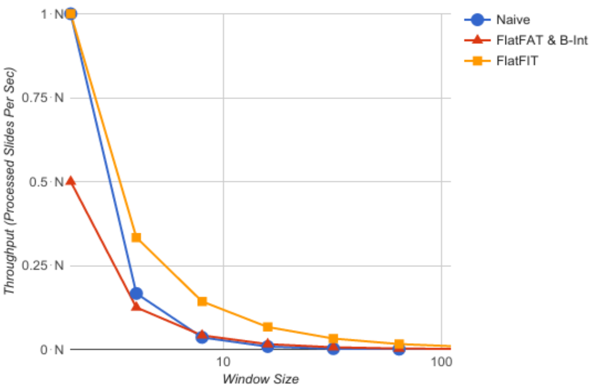

Figure 11: Theoretical throughput in a max-multi-query environment running N operations per second

4.1 Time Complexities

To compare the time complexities we calculate the number of aggregate operations needed to be executed per slide in order to return all query answers given a window size of n partial aggregates. In order to cover the entire complexity space, we provide our calculations for two boundary scenarios: a **single query** environment and a multi-query environment with maximum number of queries (which we refer to it as a **max-multi-query** environment).

In a single query environment, only one query with the range covering the entire window (which is n) is executed each slide. Such an environment can be though of as a **lower bound** of time complexity per slide. In a max-multi-query environment, the number of queries that cover all possible ranges from 1 to the length of the window (n) are executed each slide. This environment can be thought of as the **upper bound**. It is clear that the complexity of the general case (with any other numbers of queries) is between the lower and the upper bounds.

Table 1: Complexities

Algorithm	Time Complexity		Space Complexity
	Single Q	Max-Multi-Q	
Naive	$n-1$	$\frac{n^2}{2}-\frac{n}{2}$	n
FlatFAT	$log(n)$	$n \cdot log(n)$	$2^{\lceil log(n) \rceil + 1}$
B-Int	$log(n)$	$n \cdot log(n)$	$2^{\lceil log(n) \rceil + 1}$
FlatFIT	3	$n-1$	between $2n$(normal) and $2.5n$(worst)

Naive in single query environment has the time complexity of $n-1$ per each slide because it simply iterates over all the partials (n) and aggregates them. In a max-multi-query environment, at each slide $Naive$ needs to return n answers for ranges from 1 to n, which yields 0 to $n-1$ operations, respectively. By summing up this arithmetic sequence, the number of operations executed by $Naive$ per slide becomes $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$.

FlatFAT has the time complexity of $log(n)$ in a single query environment since with each new partial the binary tree is updated in a bottom-up fashion from the leaf to the root node. Since the number of levels in a binary tree is $log(n)+1$, $FlatFAT$ needs exactly $log(n)$ operations to calculate the query answer. In a max-multi-query environment it is intuitive that the upper bound of the time complexity is $n \cdot log_2 n$, since $FlatFAT$ needs to iterate over n different query ranges at each slide and each range would require $log_2 n$ operations at most to return the result. The exact *FlatFAT* complexity per slide can be produced by iterating over all possible ranges and summing their required numbers of operations together, which is equal to: $n \cdot log_2 n - \frac{3n}{2} + \frac{5log_2 n}{2} + \frac{5}{2}$. For simplicity, we use the asymptotic equivalent of this complexity which is $n \cdot log_2 n$.

B-Int similarly to *FlatFAT* is of a binary nature, and only different in how it handles updates and look-ups. It has been shown, however, to have the same time and space complexities as *FlatFAT* in [24].

FlatFIT when executed in a single query environment can be observed to execute different numbers of operations for different slides to produce the answer, however the numbers of operations follow a certain cyclical pattern which repeats every $wSize + 1$ slides.

In a single query environment the *wReset* event happens once per period. Its operational complexity with or without the optimization explained in Section 3.3 is $n - 1$ operations. *wReset* is surrounded by two slides that require just 1 operation, and the rest of the slides $(n - 1)$ in a period require two operations each. Therefore, by summing everything, we have the complexity for the natural period of *FlatFIT*: $(n - 1) + 2(n - 2) + 2 = 3(n - 1)$.

Since the above complexity is calculated for a segment of $n + 1$ slides, for a fair comparison with other approaches, we need to convert this complexity to the period of length n. To do that we multiply the above equation by n and divide by $(n - 1)$, which results in $3n$ operations for the segment of n slides, which in turn makes our complexity equal to just 3 operations per slide and is asymptotically *constant*.

In a max-multi-query environment, *FlatFIT* updates all indices at each slide by answering queries of all possible ranges, which allows it to keep the data structure maximally updated and get the answers for all queries with just one or zero operations each. In this scenario the *wReset* event happens only once at the beginning of the execution phase and is never triggered again, since the algorithm keeps all of the indices updated at all times. Due to this, at each slide *FlatFIT* returns answers to all queries in just 1 operation, except for the query with range 1, which it processes without any operations by just using the newly inserted partial value. In this scenario the operational complexity of the *FlatFIT* algorithm yields $n - 1$ operations per slide.

Takeaway Point *FlatFIT* is superior in time complexity (See Figures 8 - 11).

4.2 Space Complexities

Naive has the space complexity of n since it stores partials only once and does not keep any additional structures. This complexity stands despite the number of registered queries, since additional queries do not require any additional structures either.

FlatFAT and **B-Int** both have the space complexity of $2^{\lceil log(n) \rceil + 1}$. Because of the binary nature of these algorithms, they are more space efficient when the window size is a power of two, in which case they consume $2n$ of memory. $2n$ is made up of one n for all leaf nodes and $n - 1$ for all the tree nodes above leaves, and the first position within a flat array is normally left unused in order to simplify the addressing of nodes within the tree. In cases where the window is not a power of two, *FlatFAT* and *B-Int* round it up to the closest power of two, which is mathematically expressed as: $2^{\lceil log(n) \rceil}$. As discussed above, this complexity is multiplied by 2 in order to construct the binary data structure, yielding the final space complexity of $2^{\lceil log(n) \rceil + 1}$. Such window rounding manifests the abrupt spikes in the graph representation of these algorithms (Figure 12) and it increases algorithms' space consumption up to $3n$ in the worst cases (i.e., $n = 1025$).

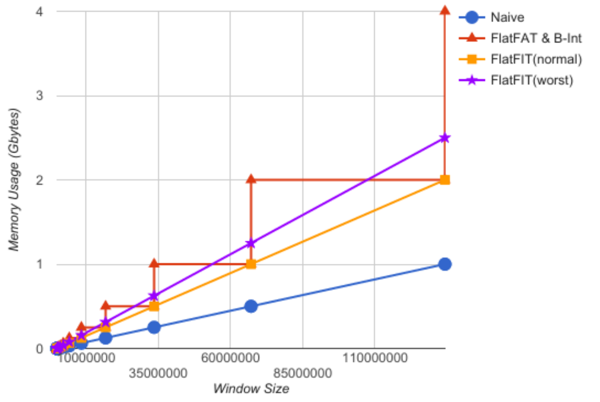

Figure 12: Theoretical Memory Usage in GB increments

FlatFIT needs two pre-allocated arrays of size n to operate and a stack that can grow up to n in size, however after introducing the optimization (in Section 3.3) in a single query environment, it cannot contain more than two values. In the max-multi-query environment the stack can contain even less: just one value at max without regard to the size of n. This makes asymptotic space complexity of *FlatFIT* $2n$. However, in terms of space complexity, single query and max-multi-query environments do not bound *FlatFIT*. In a general case where we have more than one query and less than maximum queries registered, the stack might have to store up to $n/2$ values at most, in the case with just two queries. However, each additional query (of a different range) after that cuts the maximum stack memory consumption in half by enabling higher reuse of calculations. Therefore, if the number of queries is q, the space complexity of *FlatFIT* becomes $2n$ for $q = 1$ and $q = n$, and $2n + \frac{n}{2^{q-1}}$ for the rest of the possible values of q.

Takeaway Point *Naive* is superior in space complexity, however it is clearly not feasible for heavy workloads. *FlatFIT* offers the next best space complexity while being the most scalable solution in terms of time complexity (See Figure 12).

5 EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation that confirms that the theoretical superiority of *FlatFIT* stands true in practice compared to other final aggregation approaches.

5.1 Experimental Testbed

Platform In order to test the scalability of our sliding-window aggregation processing technique, we built an experimental platform in C++ (compiled with G++5.4.1). Specifically, we implemented a stand-alone stream aggregator platform and programmed the *Naive*, *FlatFAT*, *B-Int*, and *FlatFIT* algorithms within the same

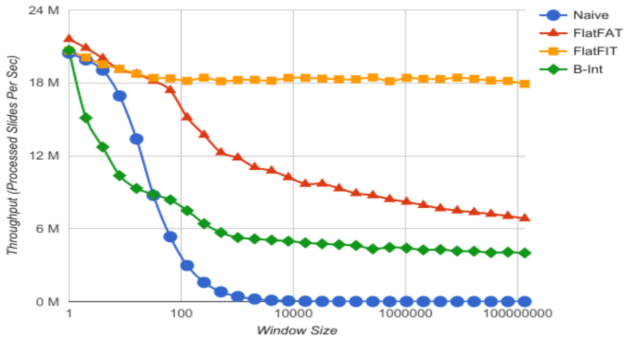

Figure 13: Throughput in processed slides per second in single query environment

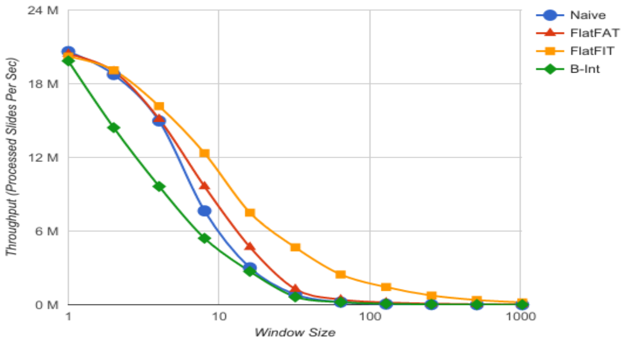

Figure 14: Throughput in processed slides per second in max-multi-query environment

codebase, sharing data structures and function calls to enable a fair comparison. Although, all the compared algorithms can be easily ported to any commercial general purpose stream processing systems, we chose to build our own stand-alone platform to carry out our evaluation in an isolated environment to avoid any potential system interferences and overheads.

Dataset We utilized the DEBS12 Grand Challenge Dataset [15], which is widely utilized in the query workload based evaluations like ours. The dataset 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 events repeating it up to ~134 million.

Workload Since the main goal of our evaluation is to compare different final aggregation techniques, we set all query slides to one tuple to reduce partial aggregation overheads to the bare minimum and measure throughput in terms of the number of query results returned per second in a single query environment. In a multi-query environment we measure how many slides of a shared execution plan each technique is able to process per second with an increasing window size, where, at each slide, multiple queries produce answers.

It is clear that the performance of the final aggregation techniques heavily depends on the window size, i.e., the larger the window size the longer it takes to process updates to it. Thus, we varied the window size from 1 tuple to 134 million tuples.

Aggregate Function For our aggregations we chose a *distributive* operation, *MAX*, as opposed to an *algebraic* operation like *MEAN* (which is decomposed into *COUNT* and *SUM* for processing) in order to benchmark the algorithms more accurately. Additionally, *MAX* is a non-invertible operation that illustrates generality of the algorithms.

5.2 Experimental Results

We ran all our experiments on an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz machine with 16 GB of RAM. For robustness, all experimental results are taken as averages of three independent runs of each experiment aggregating three different energy readings from the DEBS12 dataset.

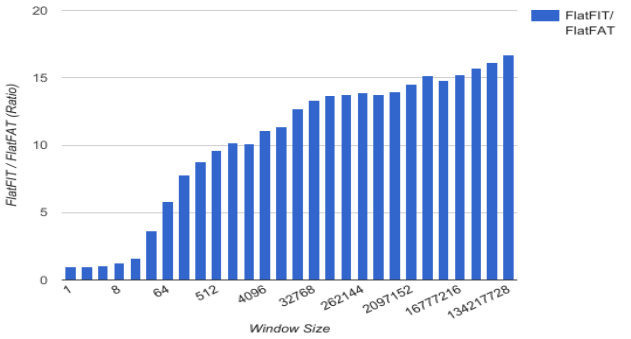

Figure 15: FlatFIT / FlatFAT throughput ratio

Exp 1: Single Query Throughput (Figure 13)
In this test we varied the window size from 1 tuple to 134 million tuples where each window is a power of two, and ran a query calculating *MAX* over the entire window after each new tuple arrival. Clearly, increasing window size increases the amount of required calculations causing lower throughputs for all four algorithms. The results are depicted in Figure 13. Notice that the rates at which throughput decreases are very similar to what we expected from the theoretical analysis of the algorithms (Figure 9).

Our statistical calculations show that *FlatFIT*'s throughputs are on average 1.8 times higher than throughputs of *FlatFAT* with a maximum of 2.6 times. We also observed that *FlatFIT* starts outperforming *FlatFAT* on windows as small as 8 tuples and increases its gain on the rest of the algorithms rapidly. *FlatFAT* showed to be more beneficial than *FlatFIT* only on window sizes from 1 to 4 tuples, however this benefit is negligible (4.4% at max).

Exp 2: Max-Multi-Query Throughput (Figures 14 and 15)
In this test we again varied the window size from 1 to 134 million tuples, however we ran a maximum number of queries calculating *MAX* value over the ranges from 1 to the window size after each new tuple arrives. In this environment, increasing window size decreases throughputs for all four algorithms much faster, because we are processing many queries per each slide, which makes the number of slides processed per second decrease quickly. The results of processing up to a window size of 1000 are depicted in Figure 14). Similarly to the previous experiment, the rates at which throughput

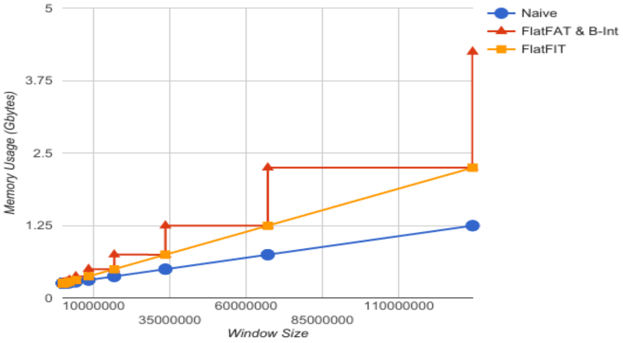

Figure 16: Experimental Memory Usage in GB increments

decreases are very similar to what we expected from the theoretical analysis of the algorithms (Figure 11). The improvement of *FlatFIT* over *FlatFAT* is depicted separately in Figure 15. Our approach demonstrated superior scalability again by yielding throughputs that are on average 10 times higher than throughputs of the state-of-the-art *FlatFAT* technique with a maximum of 17 times. Notice that in this setting *FlatFIT* performs the best on all window sizes from 2 to 134 million tuples (and only underperforms compared to *Naive* and *FlatFAT* on window size 1 by 2% and 1%, respectively).

Notice that in both experiments so far *Naive* and *FlatFAT* slightly outperformed *FlatFIT* on small window sizes (between 1 and 4 tuples). In such scenarios, the overhead of maintaining a complicated structure of *FlatFIT* outweighs the benefit of using it since the updates to the structure itself prevail the useful operation.

Exp 3: Memory Consumption (Figure 16)
In this test we again varied the window size from 1 to 134 million tuples and included window sizes that are not powers of two, and we executed a query calculating *MAX* value over the whole window size incrementally. We measured the maximum resident set size of the processes for all runs. The results of this test are depicted in Figure 16). The increasing window size increases the space requirement of the algorithms in addition to increasing the processing cost. The rates at which memory increases are almost identical to what we expected from the theoretical analysis of the space complexities (Figure 12), with only a constant difference between any two corresponding data points of all algorithms. We believe that this difference is caused by the buffering of the incoming tuples which is performed by our platform and not accounted for in the theoretical analysis. *FlatFIT* demonstrated favorable scalability again by consuming on average 1.4 times less memory than the state-of-the-art *FlatFAT* with a maximum of 1.9 times.

6 CONCLUSIONS

The main contribution of this paper is a novel technique, *FlatFIT*, for incremental sliding-window final aggregation processing. Our approach works by intelligently maintaining and reusing calculated partial aggregations in an index structure.

In the paper, we theoretically showed that *FlatFIT* significantly decreases the number of operations required for a continuous query to produce the answer while reducing the algorithm's space consumption and supporting generality in query operations. We also showed experimentally that *FlatFIT* achieves up to 2.6 times higher throughputs in a single query environment and up to 17 times in a multi-query environment compared to the current state-of-the-art algorithm. As far as we know, *FlatFIT* is the first sliding-window processing technique that has smoothed time complexity of O(1).

Acknowledgments We would like to thank Cory Thoma for his valuable discussions and insightful feedback. We would also like to thank the anonymous reviewers for their thoughtful comments.

REFERENCES

[1] Daniel J. Abadi and others. 2003. Aurora: a new model and architecture for data stream management. *VLDBJ* (2003).
[2] D. J. Abadi and others. 2005. The design of the Borealis stream processing engine. In *CIDR*.
[3] Tyler Akidau and others. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *VLDB*.
[4] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate counts and quantiles over sliding windows. In *SIGMOD*.
[5] Arvind Arasu and Jennifer Widom. 2004. Resource sharing in continuous sliding-window aggregates. In *VLDB*.
[6] Ahmet Bulut and Ambuj K Singh. 2003. SWAT: Hierarchical stream summarization in large networks. In *DataEngConf*.
[7] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*.
[8] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM journal on computing* (2002).
[9] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. 2007. Incremental Evaluation of Sliding-Window Queries over Data Streams. *TKDE* (2007).
[10] Phillip B Gibbons and Srikanta Tirthapura. 2002. Distributed streams algorithms for sliding windows. In *SPAA*.
[11] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* (1997).
[12] Shenoda Guirguis, Mohamed Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. 2012. Three-level processing of multiple aggregate continuous queries. In *ICDE*.
[13] Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. 2011. Optimized processing of multiple aggregate continuous queries. In *CIKM*.
[14] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. Streamcloud: An elastic and scalable data streaming system. *TPDS* (2012).
[15] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 grand challenge. In *DEBS*.
[16] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD*.
[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD* (2005).
[18] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*.
[19] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. 2000. Scalable algorithms for large temporal aggregation. In *DataEngConf*.
[20] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, , and Rohit Varma. 2003. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR*.
[21] Anatoli U Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2015. F1: Accelerating the Optimization of Aggregate Continuous Queries. In *CIKM*.
[22] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2015. Processing of Aggregate Continuous Queries in a Distributed Environment. In *BIRTE*.
[23] Eljas Soisalon-Soininen and Peter Widmayer. 2003. Single and Bulk Updates in Stratified Trees: An Amortized andWorst-Case Analysis. In *Computer Science in Perspective*. Springer.
[24] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *VLDB* (2015).
[25] Ankit Toshniwal and others. 2014. Storm@Twitter. In *SIGMOD*.
[26] Jun Yang and Jennifer Widom. 2001. Incremental computation and maintenance of temporal aggregates. In *DataEngConf*.