SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation

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

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

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

ABSTRACT

Online analytics, in most advanced scientific and business applications, rely heavily on the efficient execution of large numbers of Aggregate Continuous Queries (*ACQs*). Incremental sliding-window computation is used in the state-of-the-art *ACQ* processing algorithms (*FlatFIT, TwoStacks*, and *DABA*) to avoid the re-evaluation of the aggregate value of the window from scratch on every update. *FlatFIT* and *TwoStacks* aim to increase throughput, and *DABA* to minimize latency, while all process invertible and non-invertible aggregates uniformly. In this paper, we propose a novel algorithm, *SlickDeque*, that distinguishes the execution between invertible and non-invertible aggregates and offers better throughput and latency for both types. In addition, our method requires less memory and efficiently supports *multi-ACQ* processing. We theoretically show the time and space complexity advantages of *SlickDeque* and experimentally validate them using a real workload. Specifically, our approach maintains 283% lower latency spikes on average while achieving up to 19% throughput improvement in a single query environment and up to 345% improvement in a multi-query environment over the state-of-the-art approaches along with requiring up to 5 times less memory.

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 (e.g., average stock revenue, profit margin per stock, etc.) for different clients, each one with (possibly) different timing requirements. 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*) [1–3, 22, 30] have been proposed 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 streaming data, and as such they are called Aggregate Continuous Queries (*ACQs*). *ACQs* are typically associated with a *range* (**r**) and a *slide* (**s**) (also referred to as *window* and *shift* [15]), which 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.

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 answers by running aggregate queries over it. It has been shown that in sliding-window stream processing, it is beneficial to use *incremental evaluation*, which involves storing and reusing calculations performed over the unchanged parts of the window, rather than performing the re-evaluation of the entire window after each update [10, 20]. Incremental evaluation typically runs partial aggregations on the data and produces the answer by performing the final aggregation over the partial results [18, 19].

Problem Statement Handling of aggregate operations that are both *invertible* and *non-invertible* proved to be essential in domains such as finance and science. *Invertible* operations include Sum, Product, Count, Average, and Standard Deviation, while *non-invertible* operations include Max, Min, Range, Alphabetical Max (for strings), ArgMax of Cosine, and ArgMin of x^2. It was shown previously that *invertible* operations can be processed efficiently by maintaining a running Sum (or other aggregation), and invoking the inverse operation (such as Subtract) on every expiring tuple, however *non-invertible* operations require more effort to be processed efficiently and remain a challenge.

The current state-of-the-art solutions for processing *ACQs*, *FlatFIT* [26] and *TwoStacks* [28], aim to increase throughput and *DABA* [28], to minimize latency. These solutions process invertible and non-invertible aggregates uniformly, which negatively affects their performance with increasing workloads. To address the aforementioned shortcomings, in this paper we propose a novel solution named *SlickDeque*, which handles aggregate operations differently based on their invertibility property. The invertible operations are processed using *SlickDeque* (Inv), our new modified *Panes* (Inv) approach, while non-invertible *ACQs* are processed with *SlickDeque* (Non-Inv), our novel deque-based algorithm that intelligently maintains and utilizes intermediate partial aggregates allowing a greater level of reuse of previously calculated results. The separation based on invertibility leads to exceptional throughput and latency for both invertible and non-invertible operations in systems with heavy workloads.

We consider also *multi-query, multi-tenant* environments, where large numbers of *ACQs* with different ranges and slides operate on the same data stream, calculating similar aggregations.

Contributions We make the following contributions:

- We propose a novel solution for processing *ACQs*, *SlickDeque*, which processes invertible and non-invertible operations differently. *SlickDeque* is applicable for both single query and multi-query environments. (Section 3)
- We theoretically evaluate *SlickDeque* and show that it achieves better time and space complexities compared to the state-of-the-art *FlatFIT*, *TwoStacks*, and *DABA* solutions. To our knowledge, there are no prior algorithms that can achieve the same time and space complexities without loss of query generality in terms of supported aggregate operations. (Section 4)
- We experimentally evaluate *SlickDeque* based on a real dataset and show that it significantly outperforms state-of-the-art techniques in all tested scenarios by increasing the *ACQ*

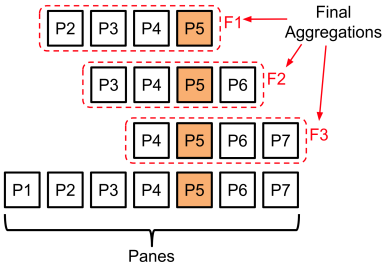

Figure 1: Panes Technique

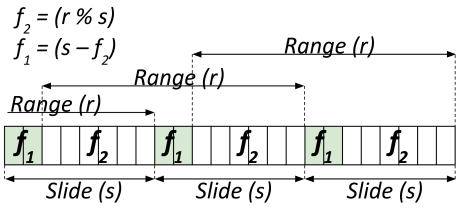

Figure 2: Paired Window Technique

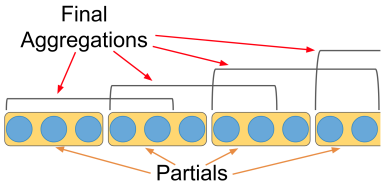

Figure 3: Cutty-slicing Technique

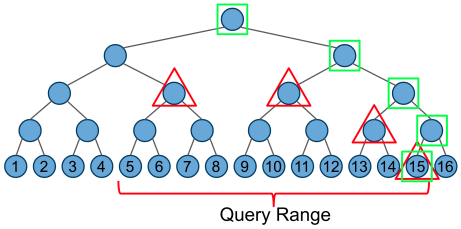

Figure 4: FlatFAT Technique

throughput by up to 19% in a single query environment and by up to 345% in a multi-query environment, while maintaining 283% lower latency spikes on average and reducing memory consumption by up to 5 times. We also show that our approach becomes superior to the state-of-the-art approaches starting at window sizes as small as eight tuples with its benefits increasing rapidly, making *SlickDeque* widely applicable for processing *ACQs* in a variety of *DSMSs*. (Section 5)

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. 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 to maintain the partials can be amortized. The following techniques aiming to reduce the number of partials were proposed for partial aggregations.

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 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 will be reused multiple times for different final aggregations. For example, in Fig. 1 partial *P5* is used 3 times as part of the final aggregations *F1*, *F2*, and *F3*.

Paired Window technique, or simply *Pairs* [18], was introduced to reduce by a factor of 2 the number of partials in a window in cases where the range is not divisible by the slide, reducing the memory consumption and accelerating the final aggregations. As illustrated in Fig. 2, two fragment lengths are used, f_1 and f_2, where $f_2 = range\%slide$ and $f_1 = slide - f_2$.

Cutty-slicing was proposed as part of the *Cutty* optimizer [8], and 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. 3). This reduces the number of partials per window by a factor of two compared to *Pairs* but it comes at a cost: additional punctuations have to be sent over the data stream to the execution module to indicate the beginnings of the new partials, which reduces the effective bandwidth of the stream and can slow down the system, especially if the workload includes a large number of queries with small windows.

2.2 Final Aggregation

The goal of final aggregation is to produce the result of a query by utilizing the partials. Initially it was performed by simply iterating over them and constructing the answer [18, 19]. For example the *Panes* technique (which we consider *Naive* in this work) in Fig. 1 performs a final aggregation *F1* 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. To improve this, several final aggregation techniques have been proposed [5, 21, 26–29, 31].

Panes (Inv) [19] (or Panes for Invertible (Differential) Aggregate Queries) was proposed to efficiently process invertible aggregates, and 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 [5] and Subtract-on-Evict [28]. In this paper we extend this approach into *SlickDeque* (Inv), which can do multi-query processing by maintaining a running aggregate for each query with a distinct range registered on the data stream.

Despite being very effective, Panes (Inv) is only applicable for invertible operations. In order to allow greater generality in query processing, the following techniques have been introduced.

FlatFAT [29] (or Flat Fixed-sized Aggregator) is a final aggregation approach which stores tuples in a pre-allocated pointer-less

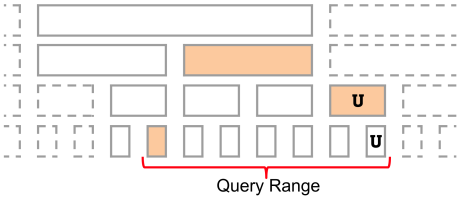

Figure 5: B-Int Technique

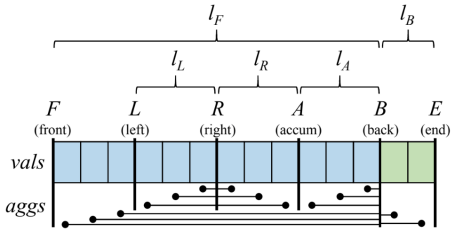

Figure 6: DABA Technique

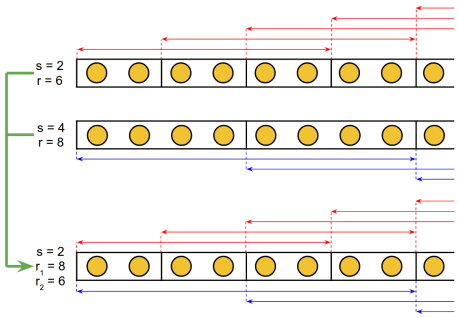

Figure 7: Shared Processing

tree-based data structure (Fig. 4), and was later extended [8] to allow partial aggregation and multi-query processing by allowing to store partial aggregates as tree leaves. Each internal node of the tree contains an aggregate of its two children. 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 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. An example of an update operation on leaf 15 is illustrated with green squares in Fig. 4. 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 leaves. The example of answering a query with a range of 11 partials starting from leaf 15 is shown with red triangles in Fig. 4.

B-Int[5] (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 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. The top level has just 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. 5). Similarly to *FlatFAT*, when producing the final aggregate, *B-Int* determines the minimum number of intervals needed to represent the desired range, and aggregates them. For example, in Fig. 5 *B-Int* aggregates all intervals marked with color to get the answer for the specified query range. Another tree-like approach similar to *FlatFAT* and *B-Int* is [6].

FlatFIT [26] (or Flat and Fast Index Traverser) was proposed with a goal of increasing the throughput of *ACQ* processing. *FlatFIT* achieves acceleration by dynamically storing the intermediate results and their corresponding pointers, which indicate how far ahead *FlatFIT* can skip in its calculation. It uses two circular arrays, *Pointers* and *Partials*, interconnected with their indices and a stack, *Positions*, for keeping indices that are currently processed. The *FlatFIT* algorithm is applicable in a multi-query environment, where it achieves a high throughput by allowing additional partial result reuse between all *ACQs* on the stream.

TwoStacks [28] was shown to also achieve a 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*. When *F* is empty, the algorithm flips *B* onto *F*, making it a calculation heavy step that introduces latency spikes to processing. To produce the final aggregation, the tops of both the *F* and *B* stacks are aggregated.

DABA [28] (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. 6 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 the pointers and fixes the consistency of the *aggs* queue.

Currently, neither *TwoStacks* nor *DABA* are known to support multi-query execution as opposed to the other above algorithms.

2.3 Shared Processing of ACQs

Since the *ACQs* are executed periodically (unlike one-shot queries), several processing schemes, as well as *ACQ* optimizers, take advantage of the shared processing of *ACQs* [8, 14, 18], 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 1 (Fig. 7) Assume two *ACQs* 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, Product, Sum, etc. and for different but compatible aggregate operations, for example Sum, Count and Average can share results by treating Average as $\frac{sum}{count}$.

To determine how many partial aggregations 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 1 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*. If slides consist of several fragments due to the partial aggregation, all fragments are also marked within the composite slide as edges. The more common edges are present in the composite slide, the more partial aggregations can be shared.

In this work we combine all compatible *ACQs* into one shared plan to achieve maximum sharing, which, in a general case, provides the most computational resource savings. Although, in specific cases it was shown that aiming for maximum sharing is not always beneficial [13, 14, 24, 25].

2.4 Other Related Work

Work similar to sliding-window aggregation exists in *Temporal Database Systems*, which store the entire stream of tuples and allow aggregations over any continuous segments of the stream, which are called *Historical Windows*. In contract, *DSMSs* generally support windows that end at or near the most recent results are referred to as *Suffix Windows* in Temporal Databases. In Temporal Databases, Red-black trees [17, 21], SB-trees, B-trees [31], and Skylines [23] are used for aggregations. 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.

Several approximate calculation approaches were proposed to save time and space by giving up accuracy [4, 7, 9, 11]. Our approach focuses solely on computing *exact* answers since it is crucial for many applications (e.g., financial, medical, etc.).

3 SLICKDEQUE

In this section we describe our new algorithm, *SlickDeque*, that significantly speeds up the final aggregation calculations in a sliding-window environment by employing different processing schemes for invertible and non-invertible aggregations.

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 [12], all aggregate operations are divided 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, e.g., Average, which is calculated from Sum and Count. The list of common distributive aggregations includes Count, Sum, Sum of Squares, Product, and Max. By combining these distributive aggregations we can calculate some commonly used algebraic aggregations 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 neither distributive nor algebraic, e.g., Median, Top-K, Quantile, Collect Distinct. 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 on optimizing the distributive aggregations; 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.
  - Note: if operation $\oplus$ is *non-invertible*, then $x \oplus y = z$, where $z \in \{x, y\}$. This is only true for **non-holistic** operations (which we target in this work).
- An operation $\oplus$ is *commutative* if $x \oplus y = y \oplus x$ is true for all x, y.

Query Operation Assumptions In terms of query operation generality, our proposed approach, *SlickDeque*, is no different from the state-of-the-art approaches, which all support non-invertible and non-commutative operations while requiring the operations 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}), and some binary operations such as *NAND* and *NOR*, are generally impractical when executed on sets of values larger than two. The difference of our proposed *SlickDeque* approach is that it has separate processing algorithms for the invertible operations (e.g., Sum, Product, Count, etc.) and non-invertible operations (e.g., Max, Min, Range, Alphabetical Max (for strings), ArgMax of Cosine, ArgMin of x^2, etc.), which allows us accelerated processing of both.

Window Structure Assumptions In *non-FIFO* window structures, the events of insertion and expiration are not synchronized, which can cause window overflow situations when there are not enough expiring tuples (or partial aggregates) to make room in the window for the insertions. All of the compared approaches, including ours, are able to handle such cases by performing dynamic resize operations. However in this paper we are focusing on the *FIFO* window environment which is the most common way to processing sliding-window aggregations in practice.

Arrival Order Assumptions Similarly, all of the aforementioned 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, some tuples fall outside of their partial, inconsistencies in the final result may arise. The mechanism that all systems uses to cope with such extreme situations is outside of the scope of this paper.

3.2 The SlickDeque Algorithm

In this subsection we provide the algorithm and implementation details for our approach followed by the clarifying examples. We

Algorithm 1 SlickDeque (Inv) Pseudocode

```
1:  Input: A set of aggregate continuous queries Q, invertible aggregate
    operation ⊕, the initial value for ⊕ initVal, the inverse operation ⊖,
    and partial aggregation technique PAT
2:  Output: Continuous answers to queries in Q according to their
    specifications.
3:                          Phase 1 (Preparation)
4:  sharedPlan = buildSharedPlan(Q, PAT)
5:  wSize = sharedPlan.wSize
6:  partials = new array[wSize]
7:  answers = new map(queryRange → answer)
8:  for i=0 to wSize do
9:      partials[i] = initVal
10: end for
11: for each query q Q do
12:     answers.insert(q.range, initVal)
13: end for
14: currPos = 0
15:                         Phase 2 (Execution)
16: while results are expected do
17:     length = sharedPlan.getNextPartialsLength()
18:     newPartial = partialAggregator.aggregate(length, PAT)
19:     for each (qR → ans) pair in answers do
20:         startPos = currPos - qR
21:         if startPos < 0 then
22:             startPos += wSize
23:         end if
24:         ans = ans ⊕ newPartial ⊖ partials[startPos]
25:     end for
26:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
27:     for each query q in queriesToAnswer do
28:         send answers.getVal(q.range) as answer to q
29:     end for
30:     partials[currPos] = newPartial
31:     currPos++
32:     if currPos == wSize then
33:         currPos = 0
34:     end if
35: end while
```

break down our algorithm description based on invertibility of the aggregate operator.

SlickDeque for Invertible Aggregates

For processing invertible aggregates we propose *SlickDeque* (Inv), a modified *Panes* (Inv) extended for processing multiple *ACQs*. Pseudocode for it is depicted in Algorithm 1. The algorithm consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase given a set of queries, Q, and one of the partial aggregation techniques (*PAT*) discussed in Section 2.1 (e.g., *Pairs*) as an input, *SlickDeque* (Inv) builds a shared execution plan by executing the *buildSharedPlan* function (line 4). The *sharedPlan* is constructed as discussed in Section 2.1, and includes a full list of partials (or edges) augmented with their lengths and lists of queries to be evaluated for each partial. The *buildSharedPlan* function identifies the query with the longest range in terms of the number of partials, and saves the range as the member *wSize* of the *sharedPlan* (line 5). *wSize* signifies the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Inv) initializes its data structures: a circular array, *partials*, (line 6) and a map, *answers*, (line 7). The *partials* array is initialized to a length equal to *wSize*, and is used to store partial aggregates. The *answers* map maintains the mappings of all queries with unique ranges to their current answers. Queries operating over the same range can share results even if they have different slides. Both the *partials* array and the values of the *answers* map are initialized (lines 8-13) with the initial value for the operation ⊕, *initVal*, supplied as input. For example, *initVal* is $-\infty$ for the Max operation.

The *currPos* variable signifies the current position within the *partials* array (line 14). 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 *currPos*.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected. At the beginning of the loop (lines 17-18), *SlickDeque* (Inv) gets the next partial's length from the *sharedPlan*, and passes it to the *newPartial Aggregator* which uses the provided *PAT* technique to produce the *newPartial* value.

Next, *SlickDeque* (Inv) loops over all range-to-answer mappings ($qR \rightarrow ans$) in the *answers* map (lines 19-25). The loop starts by identifying the start position, *startPos*, for each mapping within the *partials* array from which the values need to be aggregated. *startPos* is identified by rewinding *currPos* back by query range, *qR*, length.

Since *SlickDeque* (Inv) only works for the invertible queries, it utilizes both the aggregate operation ⊕ (e.g., Sum if query is seeking Sum), and an inverse operation ⊖ (e.g., Subtract if the original operation is Sum). This way each answer, *ans*, is updated by executing the aggregate operation ⊕ with the newly calculated *newPartial* value and the inverse operation ⊖ with expiring *partials[startPos]* value (line 24).

Next, the answers to all queries scheduled at the current position need to be produced (lines 26-29). After receiving the *queriesToAnswer* (a subset of Q) from the *sharedPlan*, *SlickDeque* (Inv) loops over them while sending back the corresponding answers pulled from the *answers* map. Then, the *Parial* value is inserted into the circular *partials* array at *currPos*, and *currPos* is moved one position forward (lines 30-34).

The following Example 2 (illustrated in Fig. 8) 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 *SlickDeque* (Inv), and we illustrate each step of their calculations side-by-side.

Example 2 Assume we have queries $Q1$ and $Q2$, which are seeking the Sum over the ranges of 3 and 5 tuples, respectively, both 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 answers to both queries need to be calculated after every new tuple arrival. Since the range of $Q2$ is 5, which is greater than the range of $Q1$, and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a *wSize* of 5 tuples.

Both *Naive* and *SlickDeque* (Inv) algorithms use the *partials* array in order to maintain incoming partial aggregates (in this case just tuples). The difference is that *Naive* produces answers to queries by iterating over this array, while *SlickDeque* (Inv) utilizes the additional *answers* map (Introduced above).

In the *partials* array we mark the positions that have been modified by the algorithm in each step. The current position (*currPos*) at each step is bolded in Fig. 8 for convenience. The tuples enter the system in the order: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization in Step 0, in Step 1 the first tuple, 6, arrives. Both algorithms store the new tuple at the *currPos* in the *partials* array, and *Naive* iterates over indexes 3, 4, and 0 in order

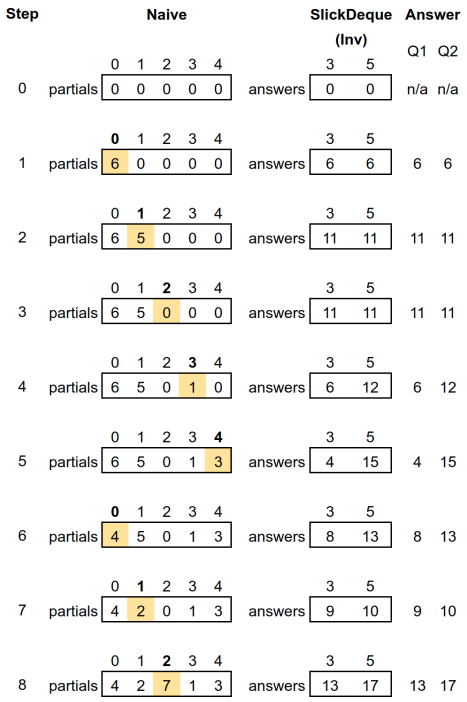

Figure 8: Example 2 processing of invertible aggregate queries Q1 and Q2 using Naive and SlickDeque (Inv) algorithms.

to answer $Q1$, and iterates over the entire array to answer $Q2$. Both answers in this case are 6.

SlickDeque (Inv) on the other hand in step 1 just updates all answers in the *answers* map by executing the operation $\oplus$ (in this example it is Sum) with the newly arrived tuple 6 and the inverse operation $\ominus$ (in this example it is Subtract) with values at indexes 2 and 0 in the *partials* array, which both are zeros. The updated answers are stored in the *answers* map.

In Step 2, the new partial, 5, arrives, and *Naive* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and sums up all of the values that were visited. The *SlickDeque* (Inv) algorithm on the other hand, is able to provide answers to both queries with just two operations each. It adds 5 and subtracts 0 from both answers in the map, making both 11.

Skipping ahead, in Step 4 *SlickDeque* (Inv) adds the new tuple, 1, to both answers, subtracts 6 from the answer to $Q1$ (since it is now out of range of $Q1$), and then subtracts 0 from the answer to $Q2$ (since 0 was in *partials[3]* in the previous step), returning 6 and 12 as answers to $Q1$ and $Q2$ respectively.

Skipping further, in Step 7 *SlickDeque* (Inv) adds 2 to both answers, and subtracts 1 from $Q1$'s answer (since it is now out of range for $Q1$) making it 9, and subtracts 5 from $Q2$'s answer (since 5 was in *partials[1]* in the previous step) making it 10. ■

Notice that in this example *Naive* had to execute a total of 48 Sum operations, while *SlickDeque* (Inv) executed a total of 32 operations (Sum and Subtract).

SlickDeque for Non-Invertible Aggregates

For processing non-invertible aggregates we propose a novel algorithm, *SlickDeque* (Non-Inv), which accelerates the processing of *ACQs* by intelligently maintaining and utilizing a deque data structure consisting of nodes allocated in chunks interconnected with pointers. For simplicity of explanation we assume that each node is allocated on a separate chunk. The benefits of allocating multiple nodes per chunk are explained in Section 4.2. Pseudocode for *SlickDeck* (Non-Inv) is depicted in Algorithm 2, and similarly to *SlickDeque* (Inv) it consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase Similarly to *SlickDeque* (Inv), the execution starts by building a *sharedPlan* by executing the function *buildSharedPlan* (line 4). It is constructed using one of the partial aggregation techniques as discussed in Section 2.1, and it includes a full list of partials augmented with their lengths and lists of queries that need to be evaluated for each partial. The query with the longest range in terms of the number of partials is identified and saved as the member *wSize* of the *sharedPlan*, signifying the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Non-Inv) defines node, *Node*, structure that has members *pos* and *val*, and initializes deque, *d*, composed of nodes, *Node*, (lines 6-7). *SlickDeque* utilizes the *currPos* variable to signify the sequential number of the current partial aggregate. It starts at 0 initially and increases to $wSize - 1$ during execution, after which it wraps back to 0.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected, and identically to *SlickDeque* (Inv), it begins by aggregating a *newPartial*. The if-statement on line 13 is removing the expired node (if present) from the head of the deque, *d*. The while-loop after that (line 16) is executing operation $\oplus$ on two values: the value of the tail node and of the new partial. If the new partial is returned by the operation, the tail node is removed from the deque (it will never be a query answer), and the next one is tested, otherwise the loop stops. The new node is then added to the deque with *currPos* as the position and *newPartial* as the value (line 19).

Next, set *queriesToAnswer* (a subset of Q scheduled at this position) is accessed from the *sharedPlan*, and the answers for its queries are produced in the for-loop below. Naturally, when the *sharedPlan* was constructed, all queries in each *queriesToAnswer* set were ordered descendingly by their range. We utilize this ordering to answer all queries by looping over the deque only once, since the larger ranges always correspond to the deque nodes closest to the head. Therefore, the position *i* within the deque is defined outside the loop and initialized to the head of the deque (line 21).

The loop starts by identifying the *startPos* of the aggregation for each query, *q*, by subtracting *q*'s range from *currPos* (line 23). If *startPos* is negative it means that this range crosses a boundary between two windows, and thus the boolean *boundaryCrossed* is set to true and *startPos* is increased by the *wSize*. Otherwise *boundaryCrossed* is set to false.

Then, based on whether the current range crosses the window boundary or not, one of the two subsequent *Answer Loops* is executed (lines 29-39), iterating over nodes from the current position *i* until the answer node is identified based on the *pos* member of each node, and returned as an answer to the query, *q*. The next iteration (to answer the next query) will continue working from the position *i* forward, until all queries are processed. After returning all required answers the *currPos* is moved one position forward (lines 42-45).

The following Example 3 (illustrated in Fig. 9) should clarify the above algorithm. To make the explanation more intuitive we again execute the two queries $Q1$ and $Q2$ on the same incoming datastream using *Naive* and *SlickDeque* (Non-Inv), and illustrate each step of their processing side-by-side.

Algorithm 2 SlickDeque (Non-Inv) Pseudocode

```
1:  Input: A set of aggregate continuous queries Q, non-invertible aggregate operation ⊕, and partial aggregation technique PAT
2:  Output: Continuous answers to queries in Q according to their specifications.
3:                          Phase 1 (Preparation)
4:  sharedPlan = buildSharedPlan(Q, PAT)
5:  wSize = sharedPlan.wSize
6:  Node with members pos and val
7:  Deque d composed of nodes of type Node
8:  currPos = 0
9:                          Phase 2 (Execution)
10: while results are expected do
11:     length = sharedPlan.getNextPartialsLength()
12:     newPartial = partialAggregator.aggregate(length, PAT)
13:     if d.size > 0 AND d.front.pos == currPos then
14:         d.pop_front()
15:     end if
16:     while d.size>0 AND d.back.val⊕newPartial==newPartial do
17:         d.pop_back()
18:     end while
19:     d.push_back(new Node(currPos, newPartial))
20:     queriesToAnswer = sharedPlan.getNextSetOfQueries()
21:     i = d.firstNode
22:     for each query q in queriesToAnswer do
23:         startPos = currPos - q.range
24:         boundaryCrossed = false
25:         if startPos < 0 then
26:             startPos += wSize
27:             boundaryCrossed = true
28:         end if
29:         if boundaryCrossed == false then
30:             //Answer Loop 1
31:             while i.pos < startPos OR i.pos > currPos do
32:                 i = i.nextNode
33:             end while
34:         else
35:             //Answer Loop 2
36:             while i.pos < startPos AND i.pos > currPos do
37:                 i = i.nextNode
38:             end while
39:         end if
40:         send i.val as answer to q
41:     end for
42:     currPos++
43:     if currPos == wSize then
44:         currPos = 0
45:     end if
46: end while
```

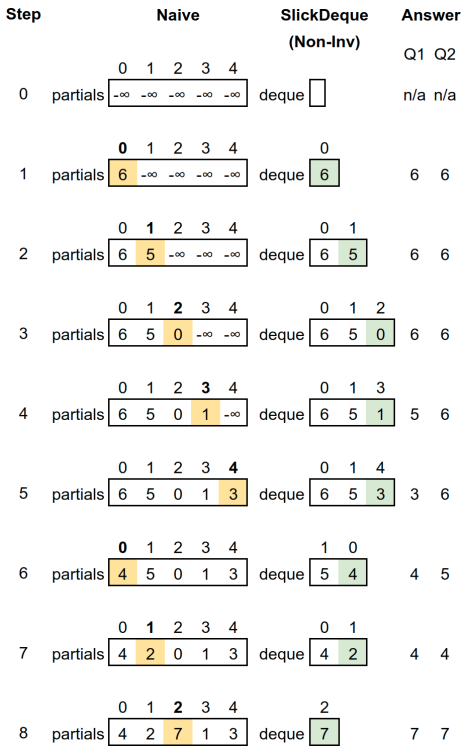

Figure 9: Example 3 processing of non-invertible aggregate queries Q1 and Q2 using Naive and SlickDeque algorithms.

Example 3 Assume we have queries $Q1$ and $Q2$, which are seeking Max over the ranges of 3 and 5 tuples respectively, both with a slide of 1 tuple. The slide size is again set to one tuple for simplicity, which means that there is no partial aggregation and the answers to both queries need to be calculated after every new tuple arrival. As before, the range of $Q2$ (5) is greater than the range of $Q1$ (3), and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a *wSize* of 5 tuples.

While *Naive* uses the circular *partials* array to maintain the incoming partials (in this case just tuples), *SlickDeque* (Non-Inv) only utilizes deque in its operation. In both *partials* and deque we mark the positions modified in each step. The tuples enter the system in the same order as in Example 2: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization Step, in Step 1 the first tuple, 6, arrives. *Naive* stores it at the *currPos* in the *partials* array, and iterates over the last 3 indexes (3, 4, and 0) to answer $Q1$, and over the entire array to answer $Q2$. Both answers in this case are 6.

SlickDeque (Non-Inv) places a new node with $pos = 0$ (which is *currPos*) and $val = 6$, at the head of the deque, and since its *pos* value is both within the last 3 and 5 positions from *currPos*, its *val* is returned as the answer to both $Q1$ and $Q2$.

In Step 2, the new partial, 5, is placed into the *currPos*, and *Naive* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and returns the Max value from all values visited, which is 6. Our algorithm on the other hand, places the new tuple 5 as a *val* of the new node (with $pos = 1$) at the end of the deque, and returns 6 (the *val* of the head node of the deque) as an answer to both queries.

Skipping ahead, in Step 4 *SlickDeque* (Non-Inv) removes the tail node of the deque since the newly arrived tuple, 1, is greater than 0, which is the *val* of the tail node, and adds the new node with $pos = 3$ and $val = 1$ at the end of the deque. Since $Q2$ has a larger range, it is scheduled to be processed first. Its *startPos* is identified: $3 - 5 = -2$, and since -2 is negative, the window boundary is crossed. Therefore *startPos* is moved to $-2+5 = 3$, and the *Answer Loop 2* is executed returning the *val* of the head node, 6. The *startPos* of $Q1$ is $3 - 3 = 0$, and since 0 is not negative, the window boundary is not crossed. Thus, the answer is produced by iterating using *Answer Loop 1*, which returned 5, the *val* of the second node from the head.

Skipping further, in Step 6 *SlickDeque* (Non-Inv) removes the head node of the deque (with $pos = 0$ and $val = 6$) which expires at this step since the *currPos* is 0. Also, since the newly arrived tuple, 4, is greater than 3, the last node of the deque is removed, and the new node with $pos = 0$ and $val = 4$ is added at the end

of the deque. $Q2$ and $Q1$ are then both processed by executing the *Answer Loop 2* and returning 5 and 4 respectively. ■

Note that this example also shows the advantage of *SlickDeque* (Non-Inv) over *Naive* by showing that *Naive* had to execute 48 Max operations total, while *SlickDeque* (Non-Inv) executed 11.

4 COMPLEXITY ANALYSIS

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

4.1 Time Complexities

We evaluate each algorithm's time complexity in terms of the number of aggregate operations it performs per slide to return all query answers given a window size of n partial aggregates. This metric was chosen because the aggregate operations are (1) applied directly to the input data, (2) constitute the the bulk of all performed operations, and (3) their number correlates best with the actual query performance. In order to cover the entire complexity space, we calculate **amortized** complexities as well as **worst-case** complexities. Amortized complexities are important to us because they correlate with *ACQ* processing throughputs, while worst-case ones reflect possible latency spikes.

In addition to providing calculations for a **single query** environment (where only one query covering the entire window is executed each slide), we also evaluate a multi-query environment with the maximum number of queries (which we refer to as a **max-multi-query** environment). This way, a single query environment can be though of as a **lower bound** of complexity per slide, while a max-multi-query environment (which executes all queries covering all possible ranges from 1 to the window length (n) each slide), can be thought of as the **upper bound**. It is clear that in most cases the complexity of the general case (with any other numbers of queries) lays between these bounds.

Naive has an exact time complexity (with matching amortized and worst cases) because it always executes the same number of operations per slide. In a single query environment, its complexity is $n-1$ (asymptotically n) because it simply iterates over all n partials and aggregates them.

In a max-multi-query environment, *Naive* needs to return n answers each slide for ranges from 1 to n, yielding 0 to $n-1$ operations, respectively. By summing up this arithmetic sequence we get $\frac{n^2}{2} - \frac{n}{2}$ (asymptotically n^2).

FlatFAT has an exact time complexity of $log_2(n)$ in a single query environment since each new partial updates the binary tree in a bottom-up fashion from the leaf to the root. Since the number of levels in a binary tree is $log_2(n)+1$, *FlatFAT* needs exactly $log_2(n)$ 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 complexity per slide can be produced by iterating over all possible ranges and summing their required numbers of operations, which equates 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: $n \cdot log(n)$.

B-Int similarly to *FlatFAT* is of a binary nature, and is only different in how it handles updates and look-ups. In [29] *B-Int* has been shown to have the same asymptotic time complexity as *FlatFAT*, with *B-Int* being slower by a constant factor, which we confirm in this work as well.

FlatFIT executes different numbers of operations for different slides, unlike *Naive*, *FlatFAT*, and *B-Int*, which causes spikes in latency. The execution of *FlatFIT* follows a cyclical pattern which repeats every $n+1$ slides, where n is the window size. In a single query environment, the so called *window reset* event happens once per such period and constitutes the worst-case complexity per slide. During the *window reset* the indexes of the entire data structure are updated in $n-1$ steps. The *window reset* operation is surrounded by two slides that require just one operation, and the rest of the slides in a period require two operations each. By summing everything, we have the amortized complexity for the natural period of *FlatFIT*: $(n-1)+2(n-2)+2 = 3(n-1)$, equating to $3n$ operations for the period of n slides, which in turn makes the amortized complexity asymptotically constant and equal to 3 operations per slide.

In a max-multi-query environment, *FlatFIT* keeps the data structure maximally updated by answering queries over all possible ranges each slide, which allows it to calculate the query answers with just one or zero operations each. Due to this, the *window reset* event happens only once at the beginning of the execution phase, and therefore in this scenario the operational complexity of the *FlatFIT* algorithm is not amortized and yields $n-1$ operations per slide (asymptotically n).

TwoStacks also executes different numbers of operations for different slides, which introduces latency spikes similarly to *FlatFIT*. During insertions, each new partial is added to the B stack and one aggregate operation is performed to determine the new aggregate value of the entire stack B. After that, another operation is performed using the top values of both the F and B stacks to return the query answer, which makes the complexity of insertions 2 operations. The majority of evictions are free since they are done by just popping the node from the F stack. When F becomes empty, however, B is flipped onto F by popping values one-by-one from B and inserting them into F while performing one aggregate operation per insertion (to populate *agg* values on F). The flip procedure (n operations) clearly constitutes the worst-case complexity per slide. To calculate the amortized complexity we add all operations per one full iteration of the algorithm: n insertions (1 operations each), n queries (1 operation each), and one eviction that causes stack flip procedure (n operations), totalling $3n$ operations per n slides. Thus, the amortized complexity of the algorithm in constant and equals 3 operations per slide. *TwoStacks* does not currently allow multi query processing.

DABA was proposed to alleviate latency spikes in *TwoStacks* by making its worst-case time complexity constant (though it still performs different numbers of operations each slide). By doing

Table 1: Algorithmic Complexities

Algorithm		Time			Space	
		Single Query		Max-Multi	Single	Max-Multi
		Amort	Worst	Query	Query	Query
Naive		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	n	$2n$	$2n$
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 $3n$.

that *DABA* sacrifices its amortized time complexity (and consequently its throughput). Per one full window iteration *DABA* executes 2 flip actions, n shift actions, and n evict actions (which all cost 0 operations), n shrink actions (costing 3 operations each), and also n insert actions and n answer look-up actions (cost 1 operation apiece), totalling $5n$ operations per n slides, which yields the amortized complexity of 5 operations. *DABA's* worst-case complexity can be attributed to a step that performs the following sequence of actions: Evict, Flip, Shrink, Insert, Shrink, Query, which costs 8 operations total. Similarly to *TwoStacks DABA* does not currently support multi query processing.

SlickDeque for Invertible Operations has an exact time complexity of just 2 operations per slide in a single query environment, since after each arrival of the new partial aggregate, the query answer is updated twice: once by executing an aggregate operation with the incoming partial, and once by executing the inverse operation with the expiring partial. In a max-multi-query environment *SlickDeque* (Inv) has to perform $2n$ operations, since one aggregate operation and one inverse operation need to be executed on each of the answers to n queries, which makes the algorithm's exact time complexity $2n$.

SlickDeque for Non-Invertible Operations executes variable numbers of operations per slide. As opposed to *FlatFIT*, *TwoStacks*, and *DABA* which are input agnostic and have their worst-case steps executed periodically, *SlickDeque* (Non-Inv) depends on the input, and the probability of ever executing its worst-case step is minuscule as we point out below.

Intuitively, in the long-running environment with a non-infinite window, each partial can causes at most two operations: one when it is inserted (invokes its comparison with the tail of the deque), and one when it is deleted by another incoming partial (invokes comparison of the incoming partial with the next item on deque). Clearly, the only two situations when a partial performs less than two operations in its lifetime are (1) if it becomes the first element of the deque after its insertion (either by removing all other partials or by being inserted into an empty deque), or (2) if it expires before being removed by another partial. If both situations happen to the same partial it will be involved in 0 operations in its lifetime. Also, it is impossible to execute a full window iteration without hitting one of the two situations by one of the partials at least once, since we cannot have an element in a deque that would both not get removed by another incoming partial as well as not expired after a full window iteration. Thus, the amortized complexity of this algorithm depends on the input, however it is always less than 2 operations.

The worst time complexity of this algorithm happens when the input (except the last partial of the window) is ordered in the opposite way of the aggregate operator order, e.g., if Max is processed and the entire input is ordered descendingly, forcing the deque to fill up, after which the next input partial has the largest value so far. This causes the new element to perform n operations while deleting all nodes on the deque. Fortunately, such a situation is highly unlikely on most inputs (1 in $n!$ chance in the uniform case). Consider the state-of-the art *DABA* algorithm that we showed to have a worst-case complexity of 8 operations. In order for *SlickDeque* (Non-Inv) to have a step with the same complexity there should be at least 9 ordered partials in the input. The probability of receiving 9 values ordered in a specific way in a row is 1 out of 9! (equals 362880), which is highly unlikely.

In a max-multi-query environment, to process all queries scheduled at a slide, the deque is traversed from the head while answering each query. Clearly, if the number of nodes in the deque is smaller than the number of different queries to answer, some nodes will have answers to multiple queries. Thus, the worst case would again be when the input forced the deque to completely fill up, for which the probability is again 1 in $n!$. In such a case, iterating over the entire deque at each step will take n operations (and at worst 2 operations per step as shown in the single query environment), so the complexity of the worst-case becomes $2n$. In the best case, the deque would have only one node each slide that would answer all queries, which would make complexity just 2 operations total.

Summary The differentiated processing of invertible and non-invertible operations allows *SlickDeque* to utilize optimizations tailored towards each type that are not available in the general case. Thus, *SlickDeque* is superior in the time complexity for both invertible and non-invertible cases compared to all other algorithms. However, in the worst-case complexity per slide, theoretically *SlickDeque* has a small possibility (1 in 362880 based on the input) to be outperformed by *DABA*.

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.

FlatFAT and **B-Int** both have the space complexity of $2^{\lceil log(n) \rceil + 1}$. Due to thir binary nature, they are more space efficient when the window size is a power of two, in which case they consume $2n$ of memory: n for all leaf nodes and $n - 1$ for all tree nodes above leaves. 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 size is not a power of two, *FlatFAT* and *B-Int* round it up to the closest power of two, which is mathematically expressed as: $2^{\lceil log(n) \rceil}$. Therefore, the space complexity of these algorithms yields $2^{\lceil log(n) \rceil + 1}$. The window rounding manifests the worst-case space complexity of $3n$.

FlatFIT needs two pre-allocated arrays of size n to operate and a stack that can grow up to 2 values total in a single query environment and in a max-multi-query environment. This results in an 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 the maximum queries registered, the stack might have to store up to $n/2$ values (case with two queries) at most. However, each additional query (of a different range) after that cuts the maximum stack memory consumption in half. 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.

TwoStacks uses stack structures with nodes containing two values, however both stacks combined can never have more than n nodes total by the nature of the algorithm, which makes its space complexity $2n$.

DABA similarly to *TwoStacks* maintains the front and back stacks with nodes consisting of both values and aggregates, however it is implemented on top of the doubly linked list of chunks. The space complexity of *DABA* depends on the number of underlying chunks, specifically, having less chunks that are bigger in size saves space on pointers (left and right), but wastes space on over-allocations (periodically window slides between chunks during

the execution leaving up to two chunks' worth of space wasted). If the window is split into k chunks, then *DABA's* space complexity is: $2n + 4k + 4n/k$. If we take a derivative with respect to k, equate it to zero, and solve for k, we conclude that the minimum space complexity for *DABA* is achieved by setting k to $\sqrt{n}$, and it equals $2n + 4\sqrt{n}$ (asymptotically $2n$).

SlickDeque for invertible operations stores partial aggregates similarly to *Naive*. In addition, it stores the answer for each query with a unique range, making its single query space complexity $n + 1$, and max-multi-query $2n$.

SlickDeque for non-invertible operations performs node allocations in chunks to reduce the space required by pointers similarly to *DABA*, causing an overallocation of up to two chunks' worth of space (at the beginning and at the end of the deque). The space complexity of *SlickDeque* (Non-Inv) does not depend on the number of registered queries, but depends on the input. In the worst-case, the input forces the deque to become full. In such a case, having n nodes with two values each, and k chunks with two pointers each, the space consumption becomes $2n+4k+4n/k$. By taking a derivative with respect to k, equating it to zero, and solving for k, we conclude that k should be set to $\sqrt{n}$ to minimize the worst-case complexity, which becomes $2n + 4\sqrt{n}$ (asymptotically $2n$). Similarly to the time complexity, the chance of the worst-case happening in normal conditions is very low: just 1 in n!. In the best case, however, each incoming partial forces the deque to eliminate all of its nodes, making the space complexity constant (2).

Summary *SlickDeque* shows a clear advantage over the rest of the algorithms in terms of space complexity. *SlickDeque* (Inv) shares the space complexity of n with *Naive*, while the rest of the algorithms have a complexity of at least $2n$, and the complexity of *SlickDeque* (Non-Inv) is always less or equal than $2n$ (based on the input). This means that only *Naive* can possibly outperform it, however the probability of that happening is low (just 1 in $n!/2$), and even then, *Naive* is still not a feasible solution because of its high time complexity.

5 EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation that confirms the theoretical superiority of *SlickDeque* in practice, by comparing it to other final aggregation approaches.

5.1 Experimental Testbed

Platform In order to test the performance of our sliding-window aggregation 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*, *FlatFIT*, *TwoStacks*, *DABA*, 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 [16] 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, which we made into a dataset of 134 million tuples.

Workload Clearly, 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, which is the maximum window size with our dataset. Given that the goal of our evaluation is just to compare different final aggregation techniques, we eliminated any side effects (i.e., overheads or benefits) induced by partial aggregation by setting all query slides to one tuple.

Evaluation Metrics We chose to compare the algorithms using throughput, latency, and memory requirement. *Throughput* is measured as the number of query results returned per second in a single query environment, while in a multi-query environment it is measured as the number of slides of a shared execution plan processed per second. *Latency* is measured in terms of the total time it took to calculate and return the answer to each query. *Memory Requirement* is measured by the maximum resident set size of processes running the corresponding techniques.

5.2 Experimental Results

We ran our experiments on an Intel(R) i7-4770 CPU @ 3.40GHz with 16 GB of RAM. For robustness, all the results were averaged over three independent runs of each experiment aggregating three different energy readings from the DEBS12 dataset.

Exp 1: Single Query Throughput

Exp1(a) Invertible Aggregates (Fig. 10)
In this experiment we varied the window size from 1 to 134 million tuples where each window is a power of two, and ran a query calculating the invertible aggregation Sum over the entire window after each new tuple arrival. From the results in Fig. 10 we clearly see that there are two groups of algorithms based on their behavior with increasing window size: (1) with constant throughput (*SlickDeque*, *FlatFIT*, *TwoStacks*, and *DABA*), and (2) with steadily degrading throughput (*FlatFAT*, *B-Int*, and *Naive*). Notice that the throughput rates are similar to what we expected from the theoretical analysis of the algorithms in Section 4.

Fig. 10 shows that *SlickDeque*'s throughput is on average 15% higher than the throughput of the second best algorithm (*FlatFAT* on windows 1 through 16, and *FlatFIT* on the rest) with a maximum of 19%. We also observed that *SlickDeque* starts outperforming other algorithms on windows as small as 4 tuples and increases its gain rapidly. *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 4 tuples, however this benefit is negligible (1% at max).

Exp1(b) Non-Invertible Aggregates (Fig. 11)
In this experiment we replaced the calculation of Sum with the non-invertible aggregation Max, that again runs over the entire window after each tuple arrival. Similarly to Exp1(a), we see that the throughput of some algorithms is practically unaffected by the increasing window size. The results are depicted in Fig. 11. Once again, the throughput rates correspond to what we expected from the theoretical analysis of the algorithms.

In this experiment *SlickDeque*'s throughput is on average 7% higher than the throughput of the second best algorithm with a maximum of 10%, and *SlickDeque* starts outperforming all other algorithms on windows as small as 16 tuples. *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 8 tuples with an advantage of 7% at max.

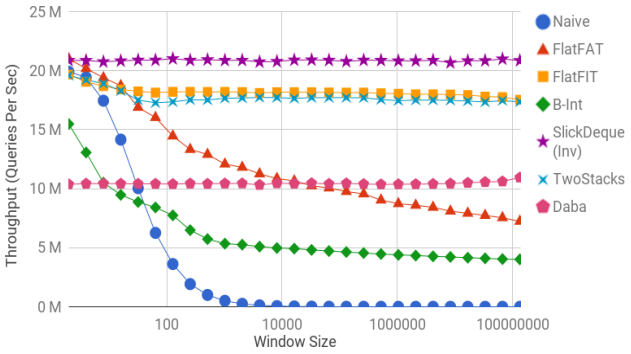

Figure 10: Exp 1 Throughput in processed queries per second in single query environment (Sum)

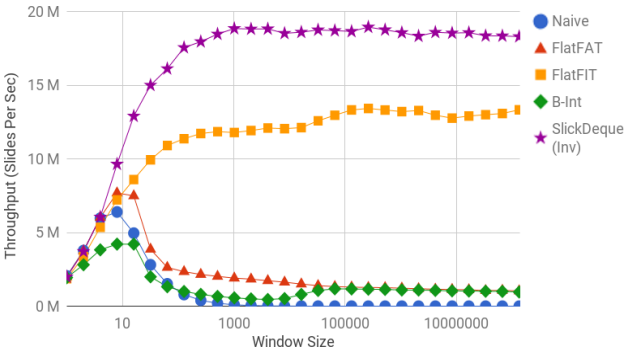

Figure 12: Exp 3 Throughput in processed slides per second in multi-query environment (Sum)

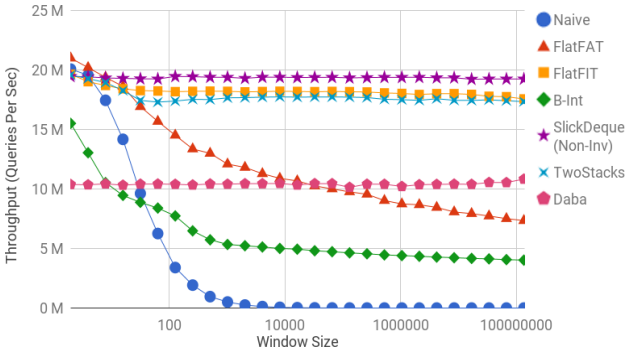

Figure 11: Exp 2 Throughput in processed queries per second in single query environment (Max)

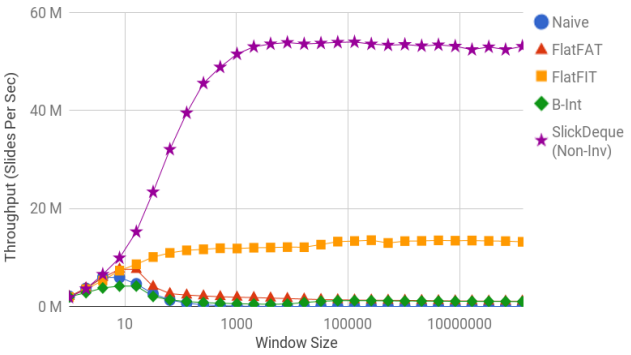

Figure 13: Exp 4 Throughput in processed slides per second in multi-query environment (Max)

Exp 2: Max-Multi-Query Throughput

Exp2(a) Invertible Aggregates (Fig. 12)
In this experiment we ran a maximum number of queries calculating Sum value over the ranges from 1 to the window size after each new tuple arrives. In this context increasing the window also increases the number of queries that are processed after each slide, enabling higher reuse of unchanged partial results among them. Thus, in Fig. 12 we see that the throughput gradually increases until the moment when the overhead of dealing with the large window outweighs the benefit of sharing between queries.

In this setting, our approach demonstrated superior scalability yet again by yielding throughput that is on average 45% higher than the throughput of the second best technique with a maximum of 60%. Notice that *SlickDeque* performs the best on window sizes from 4 tuples to 134 million tuples and only underperforms compared to other algorithms on window sizes 1 and 2 by 3% and 2%, respectively.

Exp2(b) Non-Invertible Aggregates (Fig. 13)
In this experiment we ran the maximum number of queries calculating Max over all ranges from 1 to the entire window after each tuple arrival. The results are depicted in Fig. 12, and are close to our results in experiment Exp2(a).

In this setting *SlickDeque* yielded throughput on average 266% higher than the throughput of the second best technique with a maximum of 345%. *SlickDeque* showed to perform the best on windows from 4 tuples to 134 million tuples while falling behind *Naive* and *FlatFAT* on windows 1 and 2 by 7% on average.

Summary In all throughput experiments *SlickDeque* exhibits the best results, while being slightly outperformed on small window sizes (between 1 and 8 tuples) when the overhead of maintaining its structure outweighed the benefit of using it.

Exp 3: Query Processing Latency (Fig. 14)

In this experiment we fixed our window size at 1024 tuples and ran all algorithms on the first million tuples of the DEBS data set while recording how long it took to return an answer to each query. We executed a single query processing Sum (invertible) in the first test, and Max (non-invertible) in the second test. We dropped the highest 0.005% latencies from all algorithms as outliers. The latency results of both tests were nearly identical for all algorithms except *SlickDeque*, thus we combined them in Fig. 14, where only *SlickDeque* has separate entries for invertible and non-invertible cases.

Fig. 14 shows that both invertible and non-invertible *SlickDeque* versions exhibited the lowest latency in all the following categories: Min, Max, Average, Median, 25th Percentile, and 75th Percentile. Across all of the abovementioned categories, *SlickDeque* outperformed the second best algorithm by 8% on average and 17% at most (for the non-invertible version), and by 75% average and 548% at most (for the invertible version). Also, *SlickDeque* outperformed the second best *DABA* algorithm by 283% on average in terms of the lowest max latency spike.

Exp 4: Memory Requirement (Fig. 15)

In this experiment we again varied the window size from 1 tuple to 134 million tuples (but also included window sizes that are not powers of two). We executed a query calculating the invertible Sum aggregation in the first experiment, and the non-invertible Max aggregation in the second. We measured the maximum

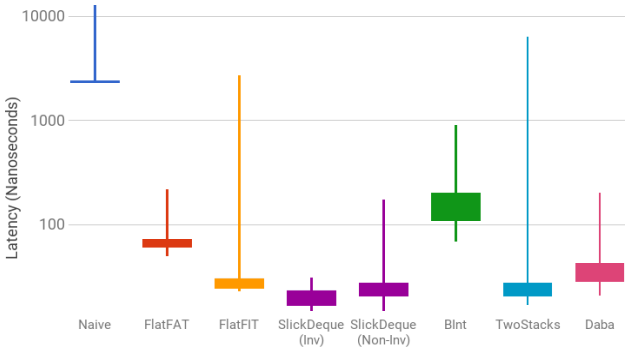

Figure 14: Exp 4 Latency in nanoseconds per query answer

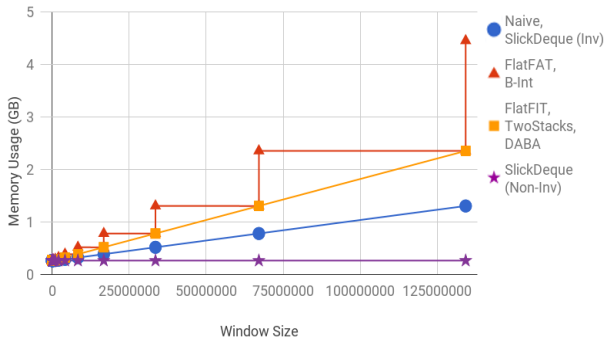

Figure 15: Exp 5 Experimental Memory Usage in Gigabyte increments

resident set size (RSS) of the processes for all runs. The results of this test are depicted in Fig. 15. On this graph, we combined the results of both invertible and non-invertible runs of all algorithms since their space requirements were identical in both Sum and Max cases except for *SlickDeque*, which we plotted separately for each case. Notice that due to the great similarity of space requirement for several algorithms, we plotted: *FlatFAT* together with *B-Int*, *FlatFIT* together with *TwoStacks* and *DABA*, *Naive* together with *SlickDeque* (Inv), and *SlickDeque* (Non-Inv) was plotted separately. The memory requirement rates correspond to what we predicted from the theoretical analysis in Section 4. *SlickDeque* demonstrated excellent scalability by matching the space usage of *Naive* for the invertible case, and for the non-invertible one outperforming the second best algorithm (*Naive*) by 2 times on average with a maximum of 5 times.

6 CONCLUSIONS

The key contribution of this paper is *SlickDeque*, a novel technique for incremental sliding-window final aggregation processing for single- and multi-query environments. Its power is the differentiated handling of aggregate operations based on their invertibility, which allows *SlickDeque* to use optimizations tailored towards each type and that are not available in the general case.

We theoretically showed that *SlickDeque* significantly decreases the number of operations required for a continuous query to return results while reducing its space requirement. As far as we know, there are no prior algorithms that can achieve the same time and space complexities without loss of query generality. We showed experimentally that *SlickDeque* achieves up to 3.5x higher throughput compared to the state-of-the-art algorithms, while maintaining up to 5.5x lower latency and utilizing up to 5x less memory. Our next step is to evaluate *SlickDeque* in dynamic and multi-node environments on production systems.

Acknowledgments Research reported in this publication was partially supported by the National Institutes of Health under Award U01HL137159 and gift from EMC. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health or EMC.

REFERENCES

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 2003.
[2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
[3] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *SIGMOD*, 2004.
[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.
[6] S. Badiozamany, K. Orsborn, and T. Risch. Framework for real-time clustering over sliding windows. In *SSDBM*, 2016.
[7] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *DataEngConf*, 2003.
[8] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, 2016.
[9] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 2002.
[10] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *TKDE*, 2007.
[11] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, 2002.
[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1997.
[13] S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, 2012.
[14] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.
[15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 2012.
[16] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *DEBS*, 2012.
[17] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *SIGMOD*, 2013.
[18] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
[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. *SIGMOD*, 2005.
[20] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.
[21] B. Moon, I. F. V. López, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *DataEngConf*, 2000.
[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
[23] D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In *SSTD*, 2017.
[24] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. F1: Accelerating the optimization of aggregate continuous queries. In *CIKM*, 2015.
[25] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Processing of aggregate continuous queries in a distributed environment. In *BIRTE*, 2015.
[26] A. U. Shein, P. K. Chrysanthis, A. Labrinidis. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*, 2017.
[27] E. Soisalon-Soininen and P. Widmayer. Single and bulk updates in stratified trees: An amortized andworst-case analysis. In *Computer Science in Perspective*. Springer, 2003.
[28] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *DEBS*, 2017.
[29] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *VLDB*, 2015.
[30] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.
[31] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *DataEngConf*, 2001.