RC25574 (WAT1511-030) November 11, 2015
Computer Science

IBM Research Report

Constant-Time Sliding Window Aggregation

Kanat Tangwongsan
Mahidol University
International College

Martin Hirzel, Scott Schneider
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Constant-Time Sliding Window Aggregation

Kanat Tangwongsan
Mahidol University
International College
kanat.tan@mahidol.edu

Martin Hirzel
IBM Research, Yorktown
Heights, NY, USA
hirzel@us.ibm.com

Scott Schneider
IBM Research, Yorktown
Heights, NY, USA
scott.a.s@us.ibm.com

ABSTRACT

Sliding-window aggregation is a widely-used approach for extracting insights from the most recent portion of a data stream. Most aggregation operations of interest can be cast as binary operators that are associative, but not necessarily commutative nor invertible. However, non-invertible operators are nontrivial to support efficiently. The best existing algorithms for this setting require $O(\log n)$ aggregation steps per window operation, where n is the window size at that point.

This paper presents DABA, a novel algorithm that significantly improves upon this time bound, assuming the sliding window has FIFO semantics. DABA requires only $O(1)$ aggregation steps, in the worst case, per window operation. As such, DABA asymptotically improves the performance of sliding-window aggregation without restricting the operator to be invertible. Furthermore, our experimental results demonstrate that these theoretic improvements hold in practice. DABA is a significant improvement over the state-of-the-art for both throughput and latency.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*systems*

General Terms

Algorithms, Performance

Keywords

Stream processing, aggregation, sliding windows, data structures, continous analytics, (de-)amortization.

1. INTRODUCTION

The last several years have witnessed a proliferation of high-speed continuous data sources in virtually all domains, including social, transportation, financial, telecommunications, medical, and more. For many of these data sources, quick reactions are more valuable than late reactions. This has led to the rise of stream processing systems. Many technology companies have built their own streaming platforms, including AT&T [8], Yahoo! [17], Microsoft [2, 16], IBM [10], Google [1], and Twitter [6, 22]. Others embrace open-source offerings [25] or purchase licenses to commercial platforms. Stream processing is growing and maturing.

Velocity is key in stream processing. In many applications, the newest data is often more relevant or valuable than older data, and has to be analyzed rapidly. To this end, most streaming systems operate on sliding windows, for example, the last hour's worth of data. The notion of sliding windows not only provides intuitive semantics to the end users but also helps bound the storage space required by a stream-processing system in a meaningful way.

Aggregation is one of the most common operations performed over sliding windows. Following Boykin et al. [6], we use the term aggregation broadly, to include both classical relational aggregation operators such as sum, average, minimum, and maximum, as well as a more general class of associative operators. For instance, Bloom filters [5] can be implemented as an associative binary operator.

For invertible operators, sliding-window aggregation is easy: one can keep a running sum and subtract off the value upon eviction. However, many aggregation operators of prime interest are not invertible. In this case, the problem becomes much more involved and has been a topic of extensive research [3, 4, 12, 13, 15, 21, 23].

To sidestep the need for an inverse operation, the state-of-the-art approaches maintain some number of partial sums in the form of a balanced aggregation tree or a set of dyadic intervals [3, 21]. This allows them to support sliding-window aggregation (*`query`*, *`insert`*, and *`evict`*) by making about $O(\log n)$ calls to the aggregation operators, where n is the window size at that time. These approaches also differ in whether the bounds are amortized or hold in the worst case.

1.1 Our Contributions

In this paper, we present novel algorithms that support sliding-window aggregation using only $O(1)$ aggregation operator invocations per sliding-window operation (*`insert`*, *`evict`*, and *`query`*), asymptotically improving upon the previous gold standard of $O(\log n)$. For constant-time aggregation operators, this ultimately means $O(1)$ time per sliding-window operation.

Our algorithms support both fixed-sized and variable-sized windows, and are designed to minimize memory allocation and pointer chasing. They work as long as the aggregation operators are associative (no requirements for invertibility or commutativity) and the window has first-in first-out (FIFO) semantics; more details of what is supported are given in Section 2. As far as we know, these are the first algorithms that are able to achieve constant-time bounds. Our main contributions are:

— *Amortized $O(1)$ Sliding-Window Aggregation:* We first describe a simple algorithm for sliding-window aggregation, called *two-stacks*, that nonetheless uses only amortized $O(1)$ aggregation-operator invocations. Our starting point is the observation that one can easily maintain aggregation on a stack (Section 3). We then derive the two-stack algorithm by extending a technique for implementing a FIFO queue using two stacks so that it supports aggregation. This is described in Section 4.

— *Worst-Case $O(1)$ Sliding-Window Aggregation:* We present an improved algorithm, called DABA, that "deamortizes" the two-stack algorithm, resulting in an algorithm for sliding-window aggregation that requires at most $O(1)$ (the constant is 3) aggregation-operator invocations per sliding-window action. We describe this algorithm in two steps. First, we present an algorithm called ABA (Section 5), which improves upon two-stacks in terms of memory allocation and data movement. Then, we derive DABA (Section 6) by systematically performing the high-latency action gradually over time.

— *Extensive Experimental Analysis:* We have implemented our new algorithms in C++ and benchmarked them against several alternative approaches. The results show that our algorithms have small overhead: only slightly slower than naïve approaches for very small windows. For moderate and large windows, they outperform existing algorithms by a large margin, thanks to the asymptotic differences. When the associative operator of the underlying aggregation (such as maximum or Bloom filters) is constant-time, then our algorithms offer constant-time sliding-window aggregation and that constant is small.

Two of our algorithms are called ABA and DABA, which stand for Amortized Banker's Aggregator and Deamortized Banker's Aggregator, respectively. The reference to banker's aggregator comes from the banker's queue of Okasaki [19], which inspired this work (see Section 8 for further discussion). The banker's queue is a persistent (functional) data structure based on the banker's method for amortized analysis of algorithms, which tracks money movements between the algorithm and a (fictitious) bank.

2. BACKGROUND AND MODEL

In this section, we formalize the problem of maintaining aggregation in a first-in first-out sliding window and discuss the kinds of aggregation operations supported in this work.

2.1 Sliding-Window Aggregation Data Type

Sliding-window aggregation is often performed on a first-in first-out (FIFO) window. In this type of window, the earliest data item to arrive is also the earliest data item to leave the window. Hence, the sliding window is essentially a queue that supports aggregation of the queue's data from the earliest to the latest. As a queue, the window is only affected by two kinds of changes:

Data Arrival: The arrival of a window data item results in a new data item at the end of the window. This is often triggered by the arrival of a data item in a relevant data stream.

Data Eviction: An eviction causes the data item at the front of the window to be removed from the window. The choice of when this happens is typically controlled by the window policy (e.g., a time based window evicts the earliest data item when it falls out of the time-frame of interest and a fixed-size window evicts the earliest data item to keep the size constant).

We model the problem of maintaining aggregation in a FIFO sliding window as an abstract data type (ADT), with an interface similar to that of a queue. To begin, we review the concept of an algebraic structure called a monoid:

Definition 1 (Monoid) A *monoid* is a triple $\mathcal{M} = (S, \oplus, \bar{\mathbb{0}})$ where $\oplus\colon S \times S \to S$ is a binary operator on S such that

– *Associativity:* For all $a, b, c \in S$, $a \oplus (b \oplus c) = (a \oplus b) \oplus c$; and

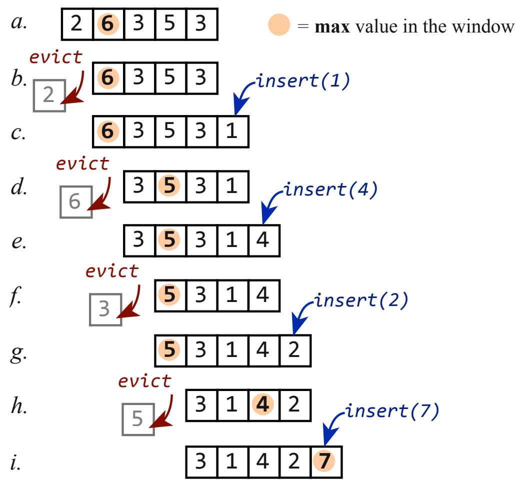

Figure 1: SWAG example trace. The sliding-window maintains the maximum value, depicted in a boldface font surrounded by a shaded circle.

– *Identity:* $\bar{\mathbb{0}} \in S$ is the identity: $\bar{\mathbb{0}} \oplus a = a = a \oplus \bar{\mathbb{0}}$ for all $a \in S$.

In comparison to real-number arithmetic, the $\oplus$ operator can be seen as a generalization of arithmetic addition where the identity element is a generalization of the number 0.

We say that a monoid is *commutative* if $a \oplus b = b \oplus a$ for all $a, b \in S$. We say that the monoid has a *left inverse* if there exists a (known and reasonably cheap) function $inv(\cdot)$ such that $a \oplus b \oplus inv(a) = b$ for all $a, b \in S$. In general, a monoid may not be commutative nor invertible.

In the context of aggregation, monoids strike a good balance between generality and efficiency as demonstrated in previous papers [6, 21, 24]. For this reason, we focus our attention on supporting monoidal aggregation, formulating the data type as follows:

Definition 2 (Sliding-Window Aggregation) The first-in first-out *sliding-window aggregation* (SWAG) data type is to maintain a collection of window data that supports the following operations:

- `new`$(\oplus, \bar{\mathbb{0}})$ creates an empty instance of SWAG that computes aggregation prescribed by the monoid that has $\oplus$ as the binary operator and $\bar{\mathbb{0}}$ as its identity element.
- `insert`(v) adds v to the rear of the sliding window. That is, if the sliding window contains values $v_0, v_2, \ldots, v_{n-1}$ in their arrival order, then `insert`(v) updates the collection to $v'_0, v'_1, \ldots, v'_n$, where $v'_i = v_i$ for $i = 0, 1, \ldots, n-1$ and $v'_n = v$.
- `evict`$()$ removes the oldest item from the sliding window. That is, if the sliding window contains values $v_0, v_1, \ldots, v_{n-1}$ in their arrival order, then `evict`$()$ updates the collection to $v'_0, v'_1, \ldots, v'_{n-2}$, where $v'_i = v_{i+1}$ for $i = 0, 1, 2, \ldots, n-2$.
- `query`$()$ returns the ordered monoidal sum of the window data. That is, if the sliding window contains values $v_0, v_1, \ldots, v_{n-1}$ in their arrival order, `query` will return $v_0 \oplus v_1 \oplus \cdots \oplus v_{n-1}$. If the window is empty, `query` will return $\bar{\mathbb{0}}$.

Throughout the paper, we will denote by n the size of the current sliding window and refer to the contents of the sliding window as $v_0, v_1, \ldots, v_{n-1}$, in their arrival order. This means v_0 is the oldest element and v_{n-1} is the youngest element.

Example: As a running example, Figure 1 shows a typical interaction with the SWAG data type. At the beginning, a SWAG instance is created with the max function as the binary operator and $-\infty$ as the identity element. It is easy to check that this is a monoid. Steps in the figure show SWAG interactions starting from a sliding window containing elements $2, 6, 3, 5, 3$. For each state in the trace, the maximum element in the window is shown in bold. Step (a)→(b) evicts the element at the front (2), causing the window to be $6, 3, 5, 3$. Step (b)→(c) then inserts 1, yielding the window $6, 3, 5, 3, 1$. The remaining steps alternate between *evict* and *insert* operations, causing the maximum to change. It should be stressed that even though in this trace, *insert* and *evict* alternate, the SWAG data type, as well all our algorithms, places no restrictions on how *insert* and *evict* may be called. They can be arbitrarily interleaved, supporting, e.g., dynamically-sized windows.

2.2 Aggregation on Monoids

Despite the simplicity, monoids are expressive enough to capture all basic aggregation operations [6, 21], as well as more sophisticated operations such as maintaining approximate membership via a Bloom filter [5], maintaining an approximate count of distinct elements [9], and maintaining the versatile count-min sketch [7].

However, many aggregation operations (e.g., standard deviation) are not themselves monoids but can be couched as operations on a monoid with the help of two extra steps. To accomplish this, prior work [21] gives a framework for the user to provide three types In, Agg, and Out and write *three* functions as follows:

- $\texttt{lift}(e : \mathsf{In}) : \mathsf{Agg}$ takes an element of the input type and "lifts" it to an element of an aggregation type that will be monoid operable.
- $\texttt{combine}(v_1 : \mathsf{Agg}, v_2 : \mathsf{Agg}) : \mathsf{Agg}$ is a binary operator that operates on the aggregation type. In our paper's terminology, *combine* is the monoidal binary operator $\oplus$.
- $\texttt{lower}(a : \mathsf{Agg}) : \mathsf{Out}$ turns an element of the aggregation type into an element of the output type.

In this framework, a query is *conceptually* answered as follows: If the sliding window consists of the elements $e_0, e_1, \ldots, e_{n-1}$, from the earliest to the latest, then *lift* derives $v_i = \texttt{lift}(e_i)$ for $i = 0, 1, 2, \ldots, n-1$. Then, *combine*, rendered as infix $\oplus$, is used to compute $a = v_0 \oplus v_1 \oplus \ldots v_{n-1}$. Finally, *lower* is used to produce the final answer as $\texttt{lower}(a)$.

Note that *lift* only needs to be applied to each element when it first arrives and *lower*, to query results at the end. As a result, the present paper focuses exclusively on the issue of maintaining the monoidal sum—i.e., how to call *combine* as rarely as possible.

3. TOOL: STACK AGGREGATION

This section describes a key ingredient of this work: how to efficiently maintain aggregation on a stack. While this may seem unrelated to sliding-window aggregation at first, it will be the basic building block for our efficient SWAG implementations.

The basic stack data type supports *push* and *pop* operations, which insert and remove an element, respectively. In a stack, the last element to arrive is the first element to be removed. Hence, both *push* and *pop* operations work on the same end. This differs from the queue, whose operations work on opposite ends. Because of this difference, aggregation on a stack is much easier to maintain.

Our solution is depicted in Figure 2. We keep a stack of inserted values (shaded in gray), where each value is associated with the (monoidal) sum of the values from the bottom of the stack to itself:

- The *push* operation has two simple steps: (1) add the element to the stack; (2) compute the corresponding sum by reading the sum from below and adding the new item to it;
- The *pop* operation is even simpler: just remove the entry at the top of the stack, including its corresponding sum; and

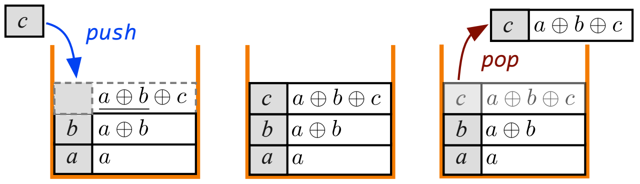

Figure 2: Aggregation on a stack-like (LIFO) collection.

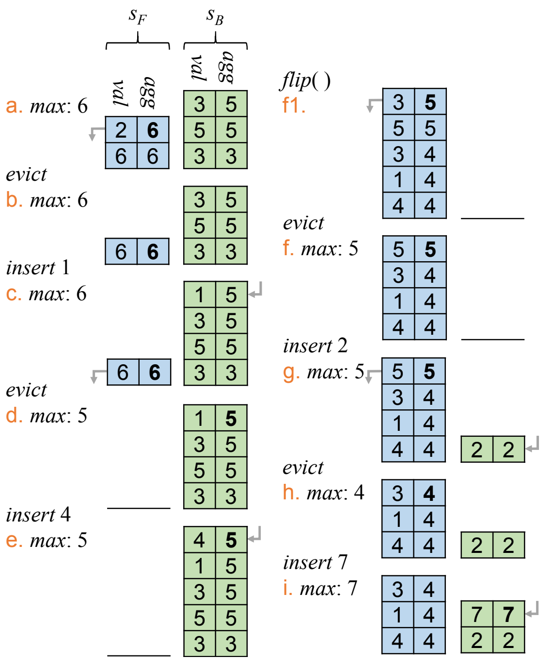

Figure 3: Two-Stacks example execution trace.

- The *query* operation simply reports the sum at the top of the stack, which equals the sum of the whole stack's contents.

Overall, this requires *one* invocation to $\oplus$ per stack operation. In other words, this solution supports *push*, *pop*, and *query* in $O(1)$ time provided that each $\oplus$ takes $O(1)$ time.

At this point, we have a stack that can maintain aggregation at virtually no additional cost. The next section describes how to use this for aggregation on a sliding window, which has the opposite data movement semantics.

4. TWO-STACKS ALGORITHM

This section presents a simple, amortized $O(1)$ algorithm that implements the SWAG data type. The main idea is to apply a classic technique from functional programming for implementing a FIFO queue using two stacks after extending it to support aggregation. The two stacks are the stack structure that supports aggregation from the previous section. The resulting algorithm invokes the monoid's $\oplus$ operation amortized $O(1)$ times per invocation of *query*, *insert*, and *evict* methods.

Example: We begin describing the algorithm by way of example. Figure 3 depicts the states of the two-stacks algorithm corresponding directly to the same states (a) through (i) in Figure 1.

In broad strokes, the two-stacks algorithm maintains the two ends of the sliding window as two separate stacks: the front stack s_F and the back stack s_B. Several features are worth noting: First, the sliding-window (FIFO) order of values can be obtained by reading

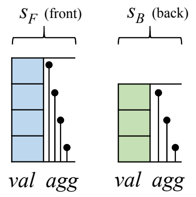

Figure 4: Two-Stacks data structure.

```
fun query()
  return $\Sigma_F^\oplus \oplus \Sigma_B^\oplus$
fun insert($v$)
  $s_B$.push(val ← $v$, agg ← $\Sigma_B^\oplus \oplus v$)
fun evict()
  if $s_F$.empty()
    flip()
  $s_F$.pop()
fun $\Sigma_F^\oplus$
  return $s_F$.empty() ? $\bar{0}$ : $s_F$.peek().agg
fun $\Sigma_B^\oplus$
  return $s_B$.empty() ? $\bar{0}$ : $s_B$.peek().agg
fun flip()
  while not $s_B$.empty()
    $v$ ← $s_B$.peek().val
    $s_B$.pop()
    $s_F$.push(val ← $v$, agg ← $v \oplus \Sigma_F^\oplus$)
```

Figure 5: Two-Stacks algorithm.

the `val` fields of the front stack s_F top-down followed by the `val` fields of the back stack s_B bottom-up.

Second, partial monoidal sums (i.e., cumulative maxima), stored in the `agg` attribute, are cumulative sums bottom-up for the respective stack. Hence, at any point, the aggregate over the entire sliding-window (i.e., the maximum value of the slding-window in this case) is the aggregation value at the top of s_F combined with the aggregation value at the top of s_B (i.e., the maximum of the two stacks in this case).

Third, `insert` pushes an element onto the back stack and `evict` pops an element from the front stack. For example, `evict` in Step (a)→(b) pops one value and cumulative sum from s_F, and `insert` in Step (b)→(c) pushes one value and cumulative sum onto s_B.

Forth, with this arrangement, it is possible to "subtract" off the evicted element without having an inverse function. Indeed, in Step (c)→(d), a new lower maximum is reached without needing the inverse of the max-monoid. To support `evict`, the algorithm simply removes the top of s_F. The problem now is, the algorithm cannot keep on popping from s_F forever; it will become empty. *What to do when the front stack s_F become empty?*

Finally, when the front stack is empty, the following `evict` reverses s_B onto s_F. Step (e)→(f) illustrates an evict on an empty s_F, which first reverses s_B onto s_F as shown in the intermediate state (f1) and proceeds with the usual evict process.

Details and Theorems: More generally, the SWAG operations can be supported as follows: Figure 4 shows the two-stacks data structure, consisting of two stacks, s_F and s_B. Focusing first on the `val` fields, the top of s_F holds the oldest value, thus optimizing for `evict`. The value at the bottom of s_F immediately precedes the value at the bottom of s_B. The top of s_B holds the newest value, thus optimizing for `insert`. In other words, s_F is in reverse order. An occasional `flip` operation reverses s_B onto s_F. Formally, the data structure obeys the invariant that the i^{th} oldest value in the FIFO is

$$v_i = \begin{cases} s_F[|s_F| - 1 - i].val & \text{if } i < |s_F| \\ s_B[i - |s_F|].val & \text{otherwise} \end{cases}$$

Turning our attention to the `agg` components, the aggregates within each of the two stacks are cumulative from the bottom. Figure 4 indicates this with the $\rceil$ notation, where `agg[•]` holds the monoidal sum of the values next to the vertical line |. Formally, the data structure obeys the invariants

$$\begin{aligned} &\forall i \in 0 \ldots |s_F| - 1 : s_F[i].agg = s_F[i].val \oplus \ldots \oplus s_F[0].val \\ \text{and}\quad &\forall i \in 0 \ldots |s_B| - 1 : s_B[i].agg = s_B[0].val \oplus \ldots \oplus s_B[i].val \end{aligned}$$

Crucially, the aggregation order in s_F is the opposite of that of s_B, so the algorithm works correctly even if $\oplus$ is not commutative. As a direct result of these invariants, the total aggregation is the sum of the aggregations at the tops of the two stacks.

Figure 5 lists the pseudo-code for the Two-Stacks algorithm. Functions `query`, `insert`, and `evict` implement the SWAG abstract data type. The remaining functions are helper functions. Functions $\Sigma_F^\oplus$ and $\Sigma_B^\oplus$ return the aggregation of the front and back stacks, respectively, as either the monoid's identity element $\bar{0}$ if the stack is empty, or the top-most `agg` otherwise. Function `flip` reverses s_B onto s_F. Since order matters for non-commutative monoids, Line 4 adds v on the right (v is younger in FIFO order), whereas Line 17 adds v on the left (v is older in FIFO order).

Theorem 3 *If the window currently contains the values $v_0, \ldots, v_{n-1}$, Two-Stacks* `query` *returns $v_0 \oplus \ldots \oplus v_{n-1}$.*

PROOF. The algorithm in Figure 5 maintains the invariants described earlier in this section. The theorem follows from those invariants. □

Theorem 4 *Each invocation of* `query`*,* `insert`*, or* `evict` *of Two-Stacks makes amortized O(1) invocations of $\oplus$.*

PROOF. The only loop occurs in `flip`, and it can be amortized by charging to the corresponding `insert` invocations that pushed elements onto s_B in the first place. □

We remark that if `query` is called more frequently than the window changes, it is useful to enhance the algorithm slightly: `query` can cache its result to avoid redundant invocations of $\oplus$. To enable this optimization, `insert` or `evict` must invalidate that cache.

One drawback of the two-stacks implementation is that function `flip` copies the entire data structure, including values. The next section shows how to avoid that copy.

5. ABA ALGORITHM

This section presents an algorithm that improves upon the two-stacks algorithm in the previous section in terms of memory efficiency. Inspired by Okasaki's notion of banker's queue, the algorithm is called *ABA* for Amortized Banker's Aggregator. Similar to the two-stacks algorithm, ABA achieves amortized $O(1)$ invocations of $\oplus$ per invocation of `query`, `insert`, and `evict`. This is also done by maintaining two stacks. However, they are maintained implicitly: ABA updates its data structure in-place and reuses memory to avoid unnecessary copying. Concepts introduced in ABA will help understand the DABA algorithm in the next section.

The ABA data structure is schematically depicted in Figure 6. The idea is as follows: Imagine taking the two stacks from the previous algorithm, rotating s_F left and s_B right, and joining the bottoms. Instead of maintaining two stacks separately, the algorithm can virtually maintain sublists l_F and l_B within a queue. Three

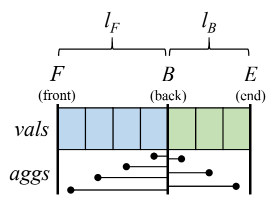

Figure 6: ABA data structure.

```
1  fun query()
2    return $\Sigma_F^{\oplus} \oplus \Sigma_B^{\oplus}$
3  fun insert($v$)
4    vals.pushBack($v$)
5    aggs.pushBack($\Sigma_B^{\oplus} \oplus v$)
6  fun evict()
7    if $F = B \wedge B \neq E$
8      flip()
9    vals.popFront(), aggs.popFront()
10 fun $\Sigma_F^{\oplus}$
11   return $(F = B)$ ? $\bar{0}$ : aggs[F]
12 fun $\Sigma_B^{\oplus}$
13   return $(B = E)$ ? $\bar{0}$ : aggs[E - 1]
14 fun flip()
15   $I \leftarrow E - 1$
16   aggs[I] $\leftarrow$ vals[I]
17   while $I \neq F$
18     $I \leftarrow I - 1$
19     aggs[I] $\leftarrow$ vals[I] $\oplus$ aggs[I + 1]
20   $B \leftarrow E$
```

Figure 7: ABA algorithm.

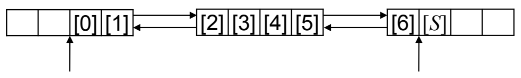

Figure 8: Chunked-array queue: the front and back pointers are marked with an arrow. S is the sentinel.

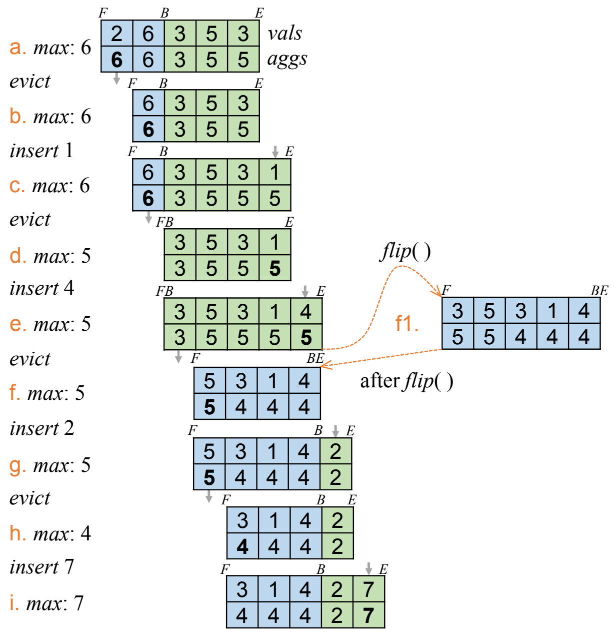

Figure 9: ABA example execution trace.

pointers F, B, and E mark the front, back, and end of the queue, respectively. These three pointers are always ordered

$$F \leqslant B \quad \text{and} \quad B \leqslant E.$$

ABA obeys the invariant that the i^{th} oldest FIFO value is

$$v_i = vals[F + i]$$

The $\bullet\!\!-$ and $-\!\!\bullet$ notation in Figure 6 indicates that $aggs[\bullet]$ holds the monoidal sum of the values above the horizontal line $-$. The corresponding invariants are

$$\begin{aligned} &\forall i \in F \ldots B-1 : aggs[i] = vals[i] \oplus \ldots \oplus vals[B-1] \\ \text{and} \quad &\forall i \in B \ldots E-1 : aggs[i] = vals[B] \oplus \ldots \oplus vals[i] \end{aligned}$$

Notice that these are the same invariants from the two-stacks algorithm, recast for the implicit-stack setting.

Low-Overhead Queue: ABA relies on an underlying queue data structure. An attractive option is a chunked-array queue, i.e., a doubly-linked list of fixed-sized arrays, as illustrated in Figure 8. Chunked-array queues implement *pushBack* and *popFront* in worst-case $O(1)$ time. Tuning the chunk size trades off allocation and dereference overhead (small chunks) against internal fragmentation (large chunks). A pointer into the chunked-array queue is a $\langle chunk, index \rangle$ pair and supports increment, decrement, read, and write all in worst-case $O(1)$ time. By placing a sentinel directly past the end of the queue, the pointer E to the end remains meaningful after *pushBack*.

Figure 7 shows the ABA algorithm. It corresponds directly to the two-stacks algorithm in Figure 5. Note that for a non-commutative monoid, the order of arguments to $\oplus$ matters (see Line 5 vs. Line 19).

Example: Figure 9 gives an example of ABA in action, using the same states (a) to (i) as the SWAG example in Figure 1. Just like for the Two-Stacks algorithm, the most interesting step for ABA is from (e) to (f) via *flip* (f1).

Theorem 5 *If the window currently contains the values $v_0, \ldots, v_{n-1}$, ABA* query *returns $v_0 \oplus \ldots \oplus v_{n-1}$.*

Proof. The algorithm in Figure 7 maintains the invariants described earlier in this section. The theorem follows from those invariants. □

Theorem 6 *Each invocation of ABA* query, insert, *or* evict *makes amortized $O(1)$ invocations of $\oplus$.*

Proof. The only loop occurs in *flip*, and it can be amortized by charging to the corresponding *insert* invocations that pushed elements onto l_B in the first place. □

The same caching optimization for *query* in the two-stacks algorithm can also be applied to *query* in ABA.

The only $O(n)$ operation of ABA is *flip*, which only occurs occasionally. Still, this is undesirable in latency-sensitive applications. The next section shows how to deamortize the algorithm, gradually carrying out the work of *flip* over time.

6. DABA ALGORITHM

Building on algorithms from the previous sections, this section describes an algorithm that supports each SWAG operation using $O(1)$ invocations of $\oplus$ *in the worst case*. The algorithm is called *DABA* for De-Amortized Banker's Aggregator, a deamortized version of ABA. We begin this section with an intuition for the ideas behind DABA (Sections 6.1 and 6.2). Following that, we present the invariants and the algorithm (Section 6.3), and conclude with theorems about correctness and complexity of DABA (Section 6.4).

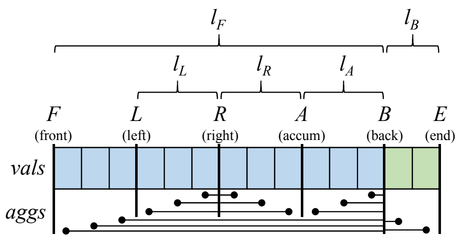

Figure 10: DABA data structure.

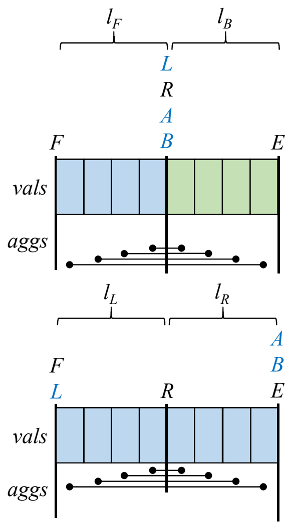

Figure 11: DABA flip.

6.1 DABA Data Structure

DABA takes ABA's data structure and enhances it with ideas to incrementally perform the `flip` operation, thereby reducing the worst-case latency by smoothing it out over a number of operations.

A few new ideas are needed. We discuss them in turn, leading to the data structure schematically depicted in Figure 10.

First, start reversing l_B early. If the algorithm waited until l_F is empty, it would be too late, as it would require a loop to reverse l_B. Instead, DABA starts earlier, and does a little bit of reversal work with each `insert` and `evict`. Observe that `insert` increases $|l_B|$ by one, and `evict` decreases $|l_F|$ by one, so both functions decrease $|l_F| - |l_B|$ by one. Therefore, DABA starts the reversal when $|l_F| = |l_B|$, and then does one unit of reversal work on each `insert` or `evict`, so it completes the reversal exactly on time.

Second, when the reversal starts, turn the old l_F and l_B into virtual sublists of the new l_F. We name the new virtual sublists l_L (left) and l_R (right). Since the reversal starts when the old l_F and l_B have equal length, the new l_L and l_R start out with equal length too. By arranging for l_L and l_R to shrink at the same pace, $|l_L| = |l_R|$ remains true throughout. Recall that in ABA, l_F and l_B can be viewed as two stacks rotated and juxtaposed at the bottoms. The boundary between the two remains fixed through normal steps of the algorithm, as illustrated in the example ABA trace in Figure 9. Similarly in DABA, the same is true for l_L and l_R: as long as they are still non-empty, while they are shrinking, the boundary between them remains fixed.

Third, leave room within l_F both before and after l_L and l_R. In the front portion of l_F, *aggs* holds partial sums up to the end of l_F, making it possible to answer `query` with $\Sigma_F^{\oplus} \oplus \Sigma_B^{\oplus}$, just like in ABA. Likewise, in the rear portion of l_F, called l_A (accumulator), *aggs* also holds partial sums up to the end of l_F.

Putting them all together, we have the data structure in Figure 10. At the top-level, it consists of two virtual sublists l_F and l_B. Within l_F, there are three virtual sublists l_L, l_R, and l_A. As before, the •– and –• notation indicates that *aggs*[•] holds the monoidal sum of the values above the horizontal line –.

6.2 DABA Incremental Reversal

The incremental reversal starts with `flip`, which merely turns the old l_F and l_B into the new l_L and l_R. Notice that `flip`, unlike before, does not contain a loop. Figure 11 illustrates that since the contents of *aggs* already have the desired layout, all this entails is assigning three pointers: `L ← F, A ← E, B ← E`.

After `flip`, the incremental reversal continues with shrinking l_L and l_R. Figure 12 illustrates this. When shrinking l_L, the first element of l_L becomes an element of the front portion of l_F. It must therefore be associated with the partial sum all the way to the B pointer. That partial sum is the sum of the aggregates of the three sublists l_L, l_R, and l_A, which are already available:

```
aggs[L] ← $\Sigma_L^{\oplus} \oplus \Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$
L ← L + 1
```

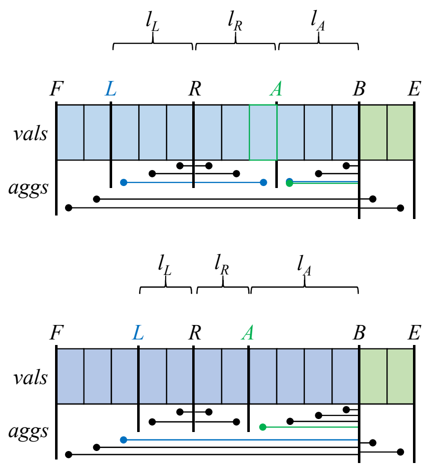

Figure 12: DABA shrinking l_L and l_R.

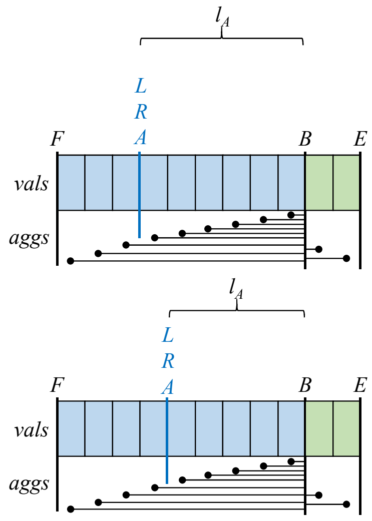

Figure 13: DABA free ride.

When shrinking l_R, the last element of l_R becomes the new first element of l_A. It must therefore extend the partial sum of l_A by one additional element:

```
aggs[A - 1] ← vals[A - 1] ⊕ $\Sigma^{\oplus}_A$
A ← A - 1
```

After the incremental reversal finishes shrinking l_L and l_R, they are empty, and what remains is the front portion of l_F followed immediately by l_A. The top of Figure 13 shows this situation. At this point, in all of l_F, the aggregates shown as •— go up to the end of l_F. That means that the algorithm can simply increment the pointers L, R, and A without changing *aggs*. In other words, this last phase of the incremental reversal is a free ride. In fact, it would even be possible to jump ahead and move pointers L, R, and A all the way to B in one fell swoop. However, incrementing them one step at a time delays the next *flip*, thus avoiding prematurely starting the next reversal.

6.3 DABA Invariants

Like ABA, DABA also obeys the invariant that the i^{th} oldest value in the FIFO is stored at

$$v_i = vals[F + i]$$

The pointers demarcating sublists are always ordered:

$$F \leqslant L \text{ and } L \leqslant R \text{ and } R \leqslant A \text{ and } A \leqslant B \text{ and } B \leqslant E$$

The invariants defining the contents of *aggs* formalize what Figure 10 shows intuitively with the •— and —• notations:

$$\begin{array}{rl} & \forall i \in F \ldots L-1 : aggs[i] = vals[i] \oplus \ldots \oplus vals[B-1] \\ \text{and} & \forall i \in L \ldots R-1 : aggs[i] = vals[i] \oplus \ldots \oplus vals[R-1] \\ \text{and} & \forall i \in R \ldots A-1 : aggs[i] = vals[R] \oplus \ldots \oplus vals[i] \\ \text{and} & \forall i \in A \ldots B-1 : aggs[i] = vals[i] \oplus \ldots \oplus vals[B-1] \\ \text{and} & \forall i \in B \ldots E-1 : aggs[i] = vals[B] \oplus \ldots \oplus vals[i] \end{array}$$

Finally, DABA has invariants on the sizes of sublists, which are central for the deamortization to work correctly:

$$\Big(|l_F| = 0 \quad \text{and} \quad |l_B| = 0\Big)$$
$$\textbf{or}$$
$$\Big(|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \quad \text{and} \quad |l_L| = |l_R|\Big)$$

The invariants treat the empty and non-empty cases separately. In the empty case ($|l_F| = 0$ and $|l_B| = 0$), only `insert` can change the data structure, since `evict` on an empty window is not allowed. After `insert`, the window has size 1, and it is trivial to arrange for that single element to be in l_F by pointer manipulation without needing the monoid's $\oplus$ operator.

The non-empty case is governed by two invariants:

- $|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B|$
  The left side of this equation is the number of incremental reversal steps required to shrink the sublists of l_F until they are empty, plus one element in the front portion of l_F to offer easy access to $\Sigma^{\oplus}_F$. The right side of this equation is the number of available steps until the next reversal must start, corresponding to the first idea in Section 6.1.
- $|l_L| = |l_R|$
  As discussed in the second idea of Section 6.1, l_L and l_R start out with the same size and then shrink at the same pace.

Figure 14 shows the entire DABA algorithm. Both `insert` and `evict` must call `fixup` to make progress on incremental reversal. Lines 21-22 handle the case where the window was empty before insert and now has exactly one element in l_B, and places that element into l_F. Lines 24-25 start the incremental reversal with a `flip`,

```
 1 fun query()
 2   return $\Sigma^{\oplus}_F \oplus \Sigma^{\oplus}_B$
 3 fun insert($v$)
 4   vals.pushBack($v$)
 5   aggs.pushBack($\Sigma^{\oplus}_B \oplus v$)
 6   fixup()
 7 fun evict()
 8   vals.popFront(), aggs.popFront()
 9   fixup()
10 fun $\Sigma^{\oplus}_F$
11   return (F = B) ? $\bar{0}$ : aggs[F]
12 fun $\Sigma^{\oplus}_B$
13   return (B = E) ? $\bar{0}$ : aggs[E - 1]
14 fun $\Sigma^{\oplus}_L$
15   return (L = R) ? $\bar{0}$ : aggs[L]
16 fun $\Sigma^{\oplus}_R$
17   return (R = A) ? $\bar{0}$ : aggs[A - 1]
18 fun $\Sigma^{\oplus}_A$
19   return (A = B) ? $\bar{0}$ : aggs[A]
20 fun fixup()
21   if F = B
22     B ← E, A ← E, R ← E, L ← E
23   else
24     if L = B
25       flip()
26     if L = R   // free ride, $l_L$ and $l_R$ are empty
27       A ← A + 1, R ← R + 1, L ← L + 1
28     else       // shrink $l_L$ and $l_R$
29       aggs[L] ← $\Sigma^{\oplus}_L \oplus \Sigma^{\oplus}_R \oplus \Sigma^{\oplus}_A$
30       L ← L + 1
31       aggs[A - 1] ← vals[A - 1] ⊕ $\Sigma^{\oplus}_A$
32       A ← A - 1
33 fun flip()
34   L ← F, A ← E, B ← E
```

Figure 14: DABA algorithm.

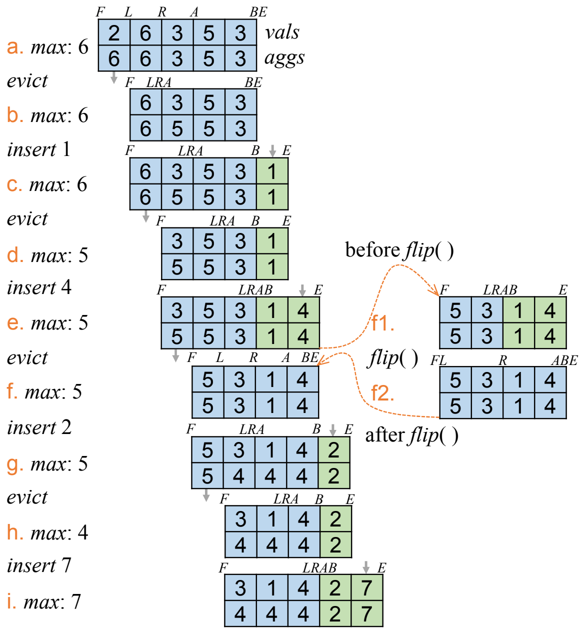

Figure 15: DABA example execution trace.

```
1  fun insert(v)
2    {$|l_F| = 0 \wedge |l_B| = 0 \vee |l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // precondition of insert: invariants from Section 6.3
3    vals.pushBack(v)
4    aggs.pushBack($\Sigma_B^{\oplus} \oplus v$)
5    {$|l_F| = 0 \wedge |l_B| = 1 \vee |l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // $l_B$ grew by 1
6    fixup()
7    {$|l_F| = 0 \wedge |l_B| = 0 \vee |l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // fixup repaired the invariants, see Line 38
8  fun evict()
9    {$|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // precondition of evict: invariants from Section 6.3, except that $l_F$ cannot be empty
10   vals.popFront(), aggs.popFront()
11   {$|l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // $l_F$ shrunk by 1
12   fixup()
13   {$|l_F| = 0 \wedge |l_B| = 0 \vee |l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // fixup repaired the invariants, see Line 38
14 fun fixup()
15   {$|l_F| = 0 \wedge |l_B| = 1 \vee |l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // disjunction of assertions before calls in Lines 5 and 11
16   if F = B
17     {$|l_F| = 0 \wedge |l_B| = 1$}    // only the first disjunct of the assertion in Line 15 can hold, the single window element is in $l_B$
18     B ← E, A ← E, R ← E, L ← E
19     {$|l_L| + |l_R| + |l_A| + 1 = |l_F| \wedge |l_B| = 0 \wedge |l_L| = 0 \wedge |l_R| = 0$}    // the single window element is now in the front portion of $l_F$
20   else
21     {$|l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // only the second disjunct of the assertion in Line 15 can hold
22     if L = B
23       {$|l_L| + |l_R| + |l_A| = 0 \wedge |l_F| = |l_B| \wedge |l_L| = 0 \wedge |l_R| = 0$}    // see "before"-picture of Figure 11 (flip)
24       flip()
25       {$|l_L| + |l_R| + |l_A| = |l_F| \wedge |l_B| = 0 \wedge |l_L| = |l_R|$}    // see postcondition of flip in Line 42 and "after"-picture of Figure 11 (flip)
26     if L = R
27       {$|l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = 0 \wedge |l_R| = 0 \wedge |l_A| > 0$}    // see "before"-picture of Figure 13 (free ride)
28       A ← A + 1, R ← R + 1, L ← L + 1
29       {$|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = 0 \wedge |l_R| = 0$}    // $l_A$ shrunk by 1, see "after"-picture of Figure 13 (free ride)
30     else
31       {$|l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R| \wedge |l_L| > 0$}    // see "before"-picture of Figure 12 (shrinking $l_L$ and $l_R$)
32       aggs[L] ← $\Sigma_L^{\oplus} \oplus \Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$
33       L ← L + 1
34       {$|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| + 1 = |l_R|$}    // $l_L$ shrunk by 1
35       aggs[A - 1] ← vals[A - 1] $\oplus\ \Sigma_A^{\oplus}$
36       A ← A - 1
37       {$|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // $l_R$ shrunk by 1, see "after"-picture of Figure 12 (shrinking $l_L$ and $l_R$)
38   {$|l_F| = 0 \wedge |l_B| = 0 \vee |l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B| \wedge |l_L| = |l_R|$}    // disjunction of Lines 19, 29, and 37; invariants from Section 6.3 hold again
39 fun flip()
40   {$|l_L| + |l_R| + |l_A| = 0 \wedge |l_F| = |l_B| \wedge |l_L| = 0 \wedge |l_R| = 0$}    // assertion before call in Line 23, see "before"-picture of Figure 11
41   L ← F, A ← E, B ← E
42   {$|l_L| + |l_R| + |l_A| = |l_F| \wedge |l_B| = 0 \wedge |l_L| = |l_R|$}    // see "after"-picture of Figure 11
```

Figure 16: Hoare-logic proof for DABA size invariants.

illustrated abstractly in Figure 11 and concretely in Step (f1)→(f2) of Figure 15. Lines 26-27 implement the free ride, illustrated abstractly in Figure 13 and concretely in several steps in Figure 15 (Steps (b)→(c), (c)→(d), (d)→(e), (g)→(h), and (h)→(i)). Finally, Lines 28-32 implement the shrinking of l_L and l_R, illustrated abstractly in Figure 12 and concretely in several steps in Figure 15 (Steps (a)→(b), (f2)→(f), and (f)→(g)).

6.4 DABA Theorems

Theorem 7 *If the window currently contains the values $v_0, \ldots, v_{n-1}$, DABA* `query` *returns $v_0 \oplus \ldots \oplus v_{n-1}$.*

Proof. The theorem follows from the invariants in Section 6.3. Most of these invariants follow directly from how the code manipulates pointers together with the corresponding *aggs* contents. Figure 16 shows a Hoare-logic proof for the size invariants. That proof relies on a corollary of the size invariants, notably, that in the non-empty case (i.e., when $\neg(|l_F| = 0 \wedge |l_B| = 0)$), the $+1$ in the size invariants guarantees $|l_F| > |l_L| + |l_R| + |l_A|$ and $|l_F| > |l_B|$. □

Theorem 8 *Each invocation of DABA* `query`*,* `insert`*, or* `evict` *makes at most $O(1)$ invocations of $\oplus$.*

Proof. The algorithm contains no loops or recursion. □

The same caching optimization for `query` in the two-stacks algorithm can also be applied to `query` in DABA. Furthermore, there is an additional caching opportunity in DABA for eliminating one of the invocations of $\oplus$ from `fixup`. Specifically, Line 29 of Figure 14 computes $\Sigma_L^{\oplus} \oplus \Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$. But this line only gets used while l_L and l_R are shrinking, and during that phase, the pointers R and B do not change. Since R and B do not change, $\Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$ does not change either, and can be cached. Figure 17 shows the variant of DABA with caching. Line 30 sets `cachedRplusA` to $\Sigma_R^{\oplus}$, and since l_A is empty at that point, that is the same as $\Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$. Line 24 uses `cachedRplusA`. There is no need to explicitly track whether the cache is valid, because it is always valid in Line 24 while R and B remain fixed.

Theorem 9 *DABA with caching invokes $\oplus$ at most one time per* `query`*, three times per* `insert`*, and two times per* `evict`*. Furthermore, for non-empty windows, DABA with caching invokes $\oplus$ on average two times per* `insert` *and one time per* `evict`*.*

Proof. The worst-case numbers can be seen directly from the highlighted invocations of $\oplus$ in Figure 17. To see the average-case numbers, consider the sequence of operations from a `flip` to the next. Immediately following `flip`, l_R is non-empty and l_A is empty. As long as l_R is non-empty, each subsequent `insert` or

```
1  fun query()
2    if not validCachedFplusB
3      cachedFplusB ← $\Sigma_F^\oplus \oplus \Sigma_B^\oplus$
4      validCachedFplusB ← true
5    return cachedFplusB
6  fun insert($v$)
7    validCachedFplusB ← false
8    vals.pushBack($v$)
9    aggs.pushBack($\Sigma_B^\oplus \oplus v$)
10   fixup()
11 fun evict()
12   validCachedFplusB ← false
13   vals.popFront(), aggs.popFront()
14   fixup()
15 fun fixup()
16   if F = B
17     B ← E, A ← E, R ← E, L ← E
18   else
19     if L = B
20       flip()
21     if L = R
22       A ← A + 1, R ← R + 1, L ← L + 1
23     else
24       aggs[L] ← $\Sigma_L^\oplus \oplus$ cachedRplusA
25       L ← L + 1
26       aggs[A - 1] ← vals[A - 1] $\oplus\ \Sigma_A^\oplus$
27       A ← A - 1
28 fun flip()
29   L ← F, A ← E, B ← E
30   cachedRplusA = $\Sigma_R^\oplus$
```

Figure 17: Caching for DABA algorithm.

`evict` operation executes Lines 24-27 of Figure 17, shrinking l_R by one and invoking $\oplus$ twice. When l_R is empty, l_A has exactly the size that l_R had at the previous `flip`. Each subsequent `insert` or `evict` operation executes Line 22 of Figure 17, shrinking l_A by one without invoking $\oplus$. The next `flip` happens when l_A is empty. That means that there was an equal number of steps shrinking l_R as l_A, and thus, an equal number of steps where `fixup` invoked $\oplus$ twice as steps where `fixup` invoked *no* $\oplus$ at all. This averages out to one $\oplus$-invocation per `fixup`, and thus, two $\oplus$-invocation per `insert` and one $\oplus$-invocation per `evict`. □

Variable-Sized Windows: DABA supports variable-sized windows. Note that Theorems 7, 8, and 9 hold irrespective of the order of insertions or evictions. Furthermore, DABA uses in-place update and simple data structures; the only memory allocation occurs when the underlying chunked-array queue grows by a chunk. Finally, DABA requires merely a monoid, whose binary operation $\oplus$ must be associative but does not need to be commutative or invertible.

This section presented DABA along with a formal evaluation of its algorithmic complexity. But given that it is a constant-time algorithm, in practice, that constant matters. That can ultimately only be evaluated empirically, which is the subject of the next section.

7. EXPERIMENTAL EVALUATION

Our experimental evaluation has three main purposes: to determine when Two-Stacks, ABA, and DABA are profitable when compared to recalculating an aggregation function over a window from scratch; to verify that our $O(1)$ theoretical result holds up in practice; and to explore their performance trade-offs in practice.

We implemented the algorithms and aggregation functions in C++11, outside of existing streaming platforms. For a particular window size n, our benchmark driver checks if the size of the SWAG is equal to n and evicts one item if it is; inserts a new data item into the SWAG; and if the size of the sliding-window aggregation (SWAG) is equal to n, queries the result. After an initial ramp-up period where the size of the SWAG grows to n, each iteration of the driver will issue an `evict`, `insert` and `query` to the SWAG.

Function	Window Size Break-Even Point			
	Reactive	Two-Stacks	ABA	DABA
Sum	1,024	48	8	64
Max	704	28	4	64
ArithmeticMean	1,024	48	28	112
GeometricMean	704	32	16	64
MinCount	512	28	16	64
SampleStdDev	704	28	16	64
ArgMax	704	28	28	48
Bloom	28	16	28	28
Collect	never	never	never	never

Table 1: Approximate break-even points. Each entry in the table is the size of the window where that aggregation algorithm started being profitable with that aggregation function, as compared to recalculating the entire window from scratch.

In our experiments, we use up to six different SWAGs: Two-Stacks, ABA, DABA with and without the caching optimization, our implementation of the Reactive Aggregator [21], and recalculating the window from scratch. Reactive serves as our comparison against current state-of-the-art; all operations on it are amortized $O(\log n)$. Recalculating the window from scratch is our performance baseline. For the rest of this section, we will refer to recalculating the window from scratch as *Recalc*.

We chose a representative sample of aggregation functions for our experiments. They range from functions that can be computed with a single instruction (Sum, Max), to functions that take many thousands of instructions (Bloom), and functions that are inherently linear (Collect). Details of these functions are provided later.

The compiler we use is g++, version 4.8.3, with the optimization level `-O3`. The operating system is CentOS 7.1, running Linux kernel version 3.10.0. The processor is an Intel Xeon 5160 at 3.0 GHz.

7.1 Break-Even Points

Table 1 answers the question, *at what window size do Two-Stacks, ABA and DABA become profitable compared to Recalc?* While we have proved that these SWAGs handle all operations in $O(1)$ time, in practice, constants matter. Table 1 shows that in practice, the break-even point is small enough for all aggregation functions, except Collect, to always use them over Recalc.

The break-even experiments look at *average* execution time, so Two-Stacks and ABA, with amortized $O(1)$ time, outperform DABA. ABA outperforms Two-Stacks because it uses the same fundamental idea but replaces two actual stacks with pointers into a single structure, which avoids unnecessary copying of data.

Reactive performs $O(\log n)$ invocations of $\oplus$, and it must maintain a tree structure, so its break-even points are between 10–100× higher than Two-Stacks, ABA and DABA. The one exception is Bloom, which is a much more expensive function.

The break-even points for some of the functions are perhaps surprising—Sum is much less expensive than Bloom, so a natural intuition is that the break-even point for Sum should be much higher. But for ABA, this is not the case. Note that these break-even points are not compared against each other, but compared against an optimized Recalc version of the function. The absolute cost of Sum is still much lower than that for Bloom, which we will explore more in the following experiments. Further, note that in practice, the cost

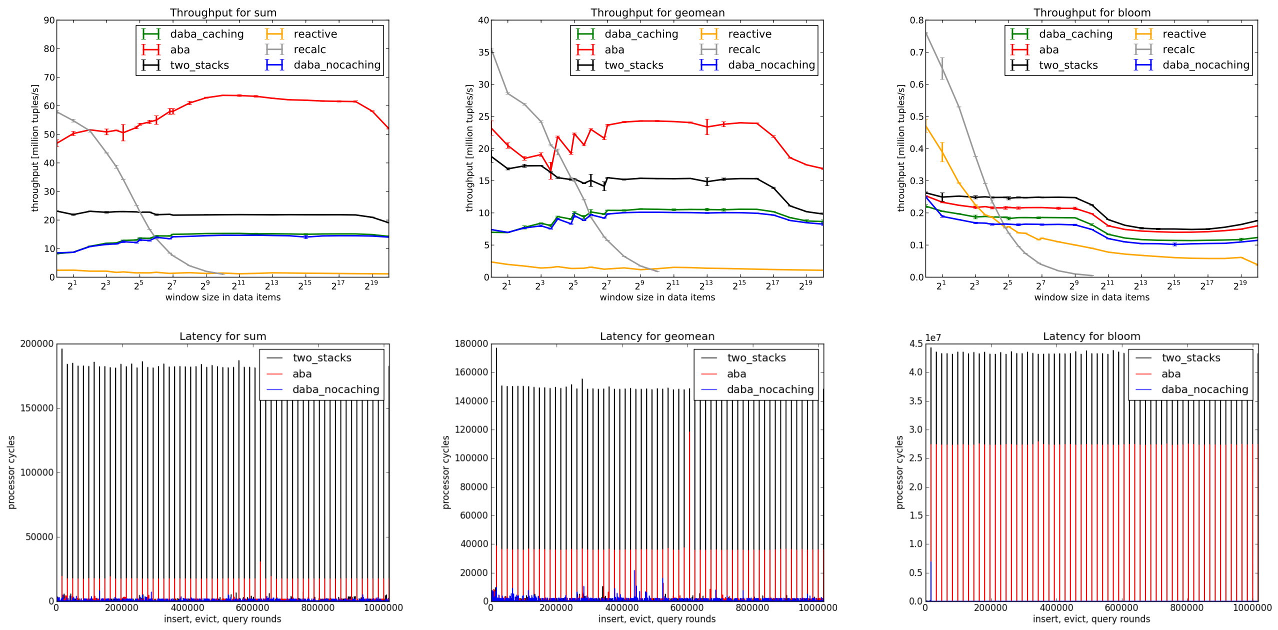

Figure 18: Top: Throughput experiments. The x-axis is the window size in number of data items in the SWAG, and the y-axis is throughput per million data items per second. **Bottom:** Latency experiments. Each graph is a single run with a window of 2^{14} data items. The x-axis is the number of rounds of `evict`, `insert`, `query` into the experiment, up to 1 million rounds. The y-axis is the number of processor cycles to execute that round.

of $\oplus$ involves not just the inherent computation itself, but also the cost of creating and returning temporary objects of type Agg. In the case of Sum, Agg is an integer. For Bloom, it is a bit-vector of size 2^{14}. Hence, for ABA, the entire cost of $\oplus$ is similar to the cost of manipulating the F, B, and E pointers. DABA needs to do more work on each invocation than both Two-Stacks and ABA, so its break-even points are necessarily higher.

The one exception is Collect, which is a special case. Collect returns the entire window as a list of size n; it is inherently $O(n)$. The Recalc version of Collect allocates one list and inserts n elements, taking $O(n)$ time. For Two-Stacks, ABA, and DABA, $\oplus$ must create a new temporary list on each invocation, and copy the elements from each operand, taking $O(n)$ time. That means all of these algorithms are linear-time, with Recalc having the lowest overheads in total.

7.2 Throughput

Our throughput experiments, top of Figure 18, explore a wide range of window sizes for three aggregation functions. We choose Sum, GeometricMean and Bloom to represent functions that are cheap, medium, and expensive.

In all three experiments, we stopped collecting data for Recalc after a window size of 2^{10}; it clearly has $O(n)$ behavior, and its throughput correspondingly trends towards 0.

The general trend is that Two-Stacks and ABA have the best throughput for all window sizes. DABA requires more bookkeeping than both Two-Stacks and ABA, so despite still having $O(1)$ behavior, its higher constant makes a measurable difference in sustained throughput. Reactive, which is the worst asymptotically after Recalc, is consistently outperformed by Two-Stacks, ABA, and DABA except for the window sizes less than 2^8 with Bloom.

For Sum and GeometricMean, ABA and DABA have the curious behavior that their performance *improves* up to a window size of 2^8. This unintuitive result is caused by the fact that the number of calls to *flip* scales down as the window size increases. Only $O(1/n)$ of window operations invoke *flip*, so the overall algorithmic complexity including flip is $O(1 + 1/n)$. Whether this behavior is clearly visible in the throughput graph depends on how costly $\oplus$ is compared to simple pointer manipulation. For Sum and GeometricMean, the manipulation of the list pointers is comparable to the cost of $\oplus$. For Bloom, $\oplus$ is substantially more expensive than manipulating the list pointers, so this effect is not noticeable.

For DABA, we experiment with the caching optimization both on and off. When the cost of copying a cached Agg result is comparable to the cost of $\oplus$, the optimization only makes a marginal difference. When $\oplus$ is more expensive, such as with Bloom, the optimization makes a modest but consistent difference in throughput.

All three functions show some degradation in performance with window sizes that approach 2^{20}, which is not predicted by the theoretical result of $O(1)$ performance. However, that result only counts invocations of $\oplus$, and does not include the effects of the memory hierarchy. This drop in performance is entirely explained by having to manage more memory. For example, a run of ABA with GeometricMean at window size of 2^{20} has about $3\times$ the number of page faults, $2\times$ the number of stalled cycles and $276\times$ the number of cache misses as a run at a window size of 2^{10}, all while having about the same number of total instructions. This pattern continues with Bloom. Sum's performance degradation happens much later because its Agg type is smaller; one 32-bit integer compared to GeometricMean's 64-bit float and 64-bit integer. This performance degradation is unavoidable: managing large window sizes will eventually cause the memory system performance to dominate the cost of a small number of $\oplus$ invocations.

7.3 Latency

All prior experiments focused on average performance, which hid the fact that Two-Stacks and ABA are amortized $O(1)$, not worst-case $O(1)$. DABA, however, is worst-case $O(1)$. We expect that periodically some calls to both Two-Stacks and ABA will be $O(n)$, whereas DABA should always be $O(1)$, keeping its latency low and well-controlled even in the worst case.

Function	Average Latency in Cycles		
	Two-Stacks	ABA	DABA
Sum	183, σ=1,428	172, σ=142	210, σ=56
Max	188, σ=1,467	160, σ=184	209, σ=58
ArithmeticMean	194, σ=1,170	171, σ=294	227, σ=69
GeometricMean	206, σ=1,169	185, σ=306	244, σ=84
SampleStdDev	218, σ=1,323	199, σ=359	255, σ=86
ArgMax	197, σ=2,047	163, σ=249	218, σ=64
Bloom	6,771, σ=338,700	7,253, σ=213,800	9,762, σ=7,957

Table 2: Average latencies and standard deviations (σ), in processor cycles, for 2^{14} data item window over 1 million rounds of `insert`, `evict`, `query`.

```
fun query()
  return agg
fun insert(v)
  vals.pushBack(v)
  agg ← agg ⊕ v
fun evict()
  v ← vals.popFront()
  agg ← agg ⊕ inv(v)
```

Figure 19: Subtract-on-evict algorithm.

```
fun query()
  agg ← 0̄
  for each v in vals
    agg ← agg ⊕ v
  return agg
fun insert(v)
  vals.pushBack(v)
fun evict()
  vals.popFront()
```

Figure 20: Recalculate-from-scratch algorithm.

The bottom of Figure 18 shows exactly this behavior. We use the same three aggregation functions, Sum, GeometricMean and Bloom. Each graph represents a latency time-series for a single run with a window size of 2^{14} data items. The x-axis is the number of `evict`, `insert`, `query` rounds into the experiment, and the y-axis is the cost in processor cycles to execute that round.

All three graphs show periodic latency spikes for Two-Stacks and ABA, where an interaction with the SWAG incurred an $O(n)$ cost. The DABA costs are all tightly bound at the bottom of the graphs. (DABA results are present for Bloom, but the cost for Two-Stacks and ABA relative to DABA makes DABA's results nearly invisible.)

The latency spikes for Two-Stacks and ABA are so high, and so frequent, that it can be difficult to reconcile the high latency in the bottom of Figure 18 with the high throughput in the top. Table 2 provides the explanation. While Two-Stacks and ABA have lower average latency than DABA, DABA's standard deviation for its latency is around 3–25× lower.

7.4 Result Summary and Discussion

As shown in the break-even and throughput experiments, Two-Stacks, ABA and DABA are a significant improvement over current state-of-the art, represented by Reactive. The latency experiments demonstrate that the tight latency bound implied by DABA's theoretic $O(1)$ cost bears out in practice.

The primary contribution of this paper is DABA's worst-case $O(1)$ guarantee. However, the experiments provide greater texture, showing that the theoretically weaker algorithms, Two-Stacks and ABA, still have merit in practice. Two-Stacks is simple to describe, understand and implement, yet is still an improvement over current state-of-the-art. ABA has worse bounds on its latency than DABA, but has higher throughput in practice. Finally, we verified DABA's tight latency bounds in practice, which makes it well-suited for environments with strict latency requirements.

8. RELATED WORK

The inspiration for DABA came from Okasaki's work on purely functional queues [18]. Purely functional data structures are implemented without any destructive modifications. Okasaki showed how to maintain a queue, as well as a deque, in worst-case constant time. Even though Okasaki's paper did not discuss aggregation, it gave us reasons to believe in the feasibility of sliding-window aggregation in worst-case constant time. Furthermore, whereas Okasaki's work relies heavily upon fine-grained memory allocation and lazy evaluation, DABA foregoes those features, achieving lower performance overhead by using destructive modifications instead.

As before, we will denote by n the window size.

A basic algorithm for sliding-window aggregation is subtract-on-evict, shown in Figure 19. It maintains a queue `vals` of values and a single variable *agg*. It requires $O(1)$ invocations of $\oplus$ for each SWAG function in the worst case. It requires $O(n)$ space. Subtract-on-evict only works when the $\oplus$ operator is invertible, whereas DABA has no such restrictions.

Another basic approach to sliding-window aggregation is recalculate-from-scratch, shown in Figure 20. It maintains a queue `vals` of values that can be walked from front to back. It requires $O(n)$ invocations of $\oplus$ for `query` and $O(n)$ space for `vals`. Like DABA, it is general, since it does not require $\oplus$ to be invertible or commutative. But the $O(n)$ `query` is only acceptable for small n.

Since subtract-on-evict is too restrictive and recalculate-from-scratch is too slow, there have been several papers on general but fast sliding-window aggregation. Most of them use some form of aggregation trees [3, 4, 15, 21, 23]. The leaves hold input values, and parent nodes combine the aggregates of their children. When the tree is balanced, it can support the SWAG functions with $O(\log n)$ invocations of $\oplus$. The difficulty is keeping the tree balanced while keeping the overhead low, especially in the presence of variable-sized windows. Like DABA, state-of-the-art tree-based algorithms for sliding window aggregation can handle non-invertible and non-commutative $\oplus$ operators. However, DABA improves upon their time complexity, from logarithmic to worst-case constant.

An orthogonal approach for speeding up sliding-window aggregation is to reduce the granularity of the window [12, 13]. The idea is to evict not individual values but batches. For instance, evict at a 1-hour granularity in a 1-day window. Then, the algorithm can pre-aggregate values that will be evicted together, thus saving not just time but also space. This means that the effective window size is reduced from n (the number of values) to b (the number of batches). It can be applied together with recalculate-from-scratch to make it $O(b)$, or with tree-based aggregation to make it $O(\log b)$, or with DABA, in which case the time complexity of $O(1)$ remains unchanged, but the space complexity improves to $O(b)$.

Finally, some literature on sliding-window aggregation focuses on sharing [3, 12]. When a system maintains multiple aggregations over the same stream that differ only in minor details (e.g., the size of the window), sharing can maintain all of them together while using less time or space than when they are maintained separately. Section 9 discusses opportunities for sharing with DABA.

9. DISCUSSION

This paper has focused on giving efficient algorithms for one incremental streaming aggregation over one FIFO window. Two-stacks and ABA achieve that with amortized constant cost, and DABA achieves that with worst-case constant cost. However, some use cases call for several streaming aggregations differing only in the window size or the monoid. The question is, whether this can be accomplished with less time by sharing computations or with less space by sharing data structures. Furthermore, some use cases call for windows that are not strictly FIFO, because data items carry timestamps inconsistent with their arrival order. The question is, whether the window can still be aggregated and evicted correctly.

Sharing: It helps to observe that once a value is inserted into `vals`, it remains unchanged until it gets evicted. That means, two aggregations over the *same window size but with different monoids* can share a single `vals` queue. They merely need to maintain separate `aggs` queues. If the monoids are closely related, they can share even more. For instance, an arithmetic-mean aggregation can reuse the `aggs` queue of a sum if sum is also being computed. Next, we consider the case of *different window sizes but with the same monoid*. In this case, it is possible to maintain the `vals` queue for the largest window only, and share it. Furthermore, if there are only a few different window sizes, they can share even more. For instance, to aggregate both a 1-day window and a 2-day window, keep two DABAs, *young* and *old*. Insert new data items into *young*. When a datum becomes 1 day old, evict it from *young* and insert it into *old*. Finally, the sharing techniques above can be combined to handle *different window sizes with different monoids*. More elaborate sharing is left to future work, and may take inspiration from the literature [3, 12].

Non-FIFO Windows: DABA can be applied together with techniques from the literature on out-of-order streaming aggregation. Assuming there is a *latency bound* on how late data items arrive, we suggest a variation on Srivastava and Widom's approach [20]. The idea is to aggregate data younger than the latency bound in a balanced tree ordered by timestamp, and data older than the latency bound in DABA. A data item that reaches the latency bound is evicted from the tree and inserted into DABA. Assuming there are independent data sources that are internally FIFO but can have large and *unbounded drift* with respect to each other, we suggest a variation on Krishnamurthy et al.'s approach [11]. For this case, we shall assume that the monoid is not just associative but also commutative—a reasonable assumption since the aggregation needs to be order-agnostic. We can keep the aggregation over each source in its own independent DABA. Then, we can combine all the DABAs using a balanced tree, whose leaves are individual DABAs and whose root yields the overall aggregation. More elaborate handling of non-FIFO windows is left to future work, and may take inspiration from the literature [11, 14, 20].

To summarize, DABA enables some simple forms of sharing, and could be combined naturally with some prior work on out-of-order stream processing.

10. CONCLUSION

This paper presented DABA, a new algorithm for incremental sliding-window aggregation. DABA can maintain aggregation for any monoid, using its binary associative operator $\oplus$ to aggregate the window contents. Intuitively, it works as follows. DABA maintains partial monoidal sums over sublists of the values in the window, such that *`query`* can easily derive the complete monoidal sum any time. DABA ensures that upon an *`insert`*, the partial sum for the newly-inserted slot is easy to compute. It also ensures that upon an *`evict`*, the partial sum for the next-to-oldest slot is easy to compute. Most importantly, DABA incrementally fixes the sublist boundaries during *`insert`* and *`evict`* to avoid ever having to perform a large number of steps during a window update.

DABA has several desirable properties: it only requires an associative monoid (no need for commutativity nor invertibility). DABA is the first sliding-window aggregation algorithm that only requires $O(1)$ invocations of $\oplus$ for each *`insert`*, *`evict`*, or *`query`* invocation, irrespective of the current window size. More specifically, the worst-case is three invocations of $\oplus$. DABA uses $O(n)$ space, where n is the window size. DABA supports dynamically-sized windows, where the window size fluctuates throughout the execution, for instance, due to a variable inter-arrival rate of stream data items. DABA is built on a simple flat data structure, thus avoiding memory-copy or allocation churn, as well as avoiding excessive pointer chasing. Our experiments demonstrate that an implementation of DABA performs well compared to other incremental sliding-window aggregation algorithms.

References

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases (VLDB) Industrial Track*, pages 734–746, 2013.

[2] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in Microsoft StreamInsight. In *International Conference on Data Engineering (ICDE)*, pages 1242–1253, 2011.

[3] A. Arasu and J. Widom. Resource sharing in continuous sliding window aggregates. In *Conference on Very Large Data Bases (VLDB)*, pages 336–347, 2004.

[4] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental sliding window analytics. In *International Conference on Middleware*, pages 61–72, 2014.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM (CACM)*, 13(7), 1970.

[6] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online MapReduce computations. In *Conference on Very Large Data Bases (VLDB)*, pages 1441–1451, 2014.

[7] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *International Conference on Management of Data (SIGMOD) Industrial Track*, pages 647–651, 2003.

[9] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms (AofA)*, pages 127–146, 2007.

[10] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development (IBMRD)*, 57(3/4), 2013.

[11] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *International Conference on Management of Data (SIGMOD)*, pages 1081–1092, 2010.

[12] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *International Conference on Management of Data (SIGMOD)*, pages 623–634, 2006.

[13] 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. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[14] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. In *Conference on Very Large Data Bases (VLDB)*, pages 274–288, 2008.

[15] B. Moon, I. F. V. López, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *International Conference on Data Engineering (ICDE)*, pages 145–154, 2000.

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream processing platform. In *Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*, pages 170–177, 2010.

[18] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming (JFP)*, 5(4):583–592, 1995.

[19] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[20] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Principles of Database Systems (PODS)*, pages 263–274, 2004.

[21] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. In *Conference on Very Large Data Bases (VLDB)*, pages 702–713, 2015.

[22] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @Twitter. In *International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[23] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *International Conference on Data Engineering (ICDE)*, pages 51–60, 2001.

[24] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.