Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time

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

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Scott Schneider
IBM Research
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. The aggregations of interest can usually be cast as binary operators that are associative, but they are not necessarily commutative nor invertible. Non-invertible operators, however, are difficult to support efficiently. The best published algorithms require $O(\log n)$ aggregation steps per window operation, where n is the sliding-window size at that point. For a FIFO window, this can be improved to $O(1)$ on average by using two aggregation stacks.

This paper presents DABA, a novel algorithm for aggregating FIFO sliding windows that significantly improves upon these time bounds. DABA requires only $O(1)$ aggregation steps per operation in the worst case (not just on average). As such, DABA asymptotically improves the performance of sliding-window aggregation without restricting the operator to be invertible. Our experimental results demonstrate that these theoretical improvements hold in practice. DABA is a substantial improvement over the state of the art in terms of both latency and throughput.

CCS Concepts •**Information systems → Stream management;**

Keywords Real-time, continuous analytics, (de-)amortization

ACM Reference format:
Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 13 pages.
DOI: 10.1145/3093742.3093925

1 Introduction

Stream processing is a now-standard paradigm for handling high-speed continuous data, spurring the development of many stream-processing engines in the past few years [1, 3, 4, 8, 10, 13, 18, 21, 28, 31]. Since stream processing is often subject to strict quality-of-service requirements or real-time requirements, low-latency responses are often a must. Aggregation (e.g., computing the sum or geometric mean) is a mainstay of stream processing. It is one of the most common computations in streaming applications, used both standalone and as a building block for more sophisticated analytics. Unfortunately, existing techniques for sliding-window aggregation cannot consistently guarantee low latency.

Because the newest data is often deemed more pertinent or valuable than older data, streaming aggregation is typically performed on a sliding window (e.g., the last hour's worth of data). This not only provides intuitive semantics to the end users but also helps bound the amount of data the system has to keep around. An algorithm for sliding-window aggregation supports three operations (formally described in Section 2): `insert` for a data item's arrival, `query` for requesting the current aggregation outcome, and `evict` for a data item's departure. Following Boykin et al. [8], we use the term aggregation broadly, to include both classical relational aggregation operators such as sum, geometric mean, and maximum, as well as a more general class of associative operators. For instance, Bloom filters [7] can be implemented as an associative binary operator.

Algorithm	Time	Space	Invertible	FIFO
Subtract on Evict	worst $O(1)$	$O(n)$	needed	no
Recalculate from Scratch	worst $O(n)$	$O(n)$	no	no
Reactive Aggregator [27]	avg. $O(\log n)$	$O(n)$	no	no
Two-Stacks	avg. $O(1)$	$O(n)$	no	needed
FOA and IOA	worst $O(1)$	$O(n)$	no	needed
DABA	worst $O(1)$	$O(n)$	no	needed

Table 1. Comparison of sliding window aggregation algorithms. Time and space indicate algorithmic complexity. Invertible indicates a limitation on $\oplus$, and FIFO indicates a limitation on the window. For example, Subtract on Evict runs in worst-case $O(1)$ time, uses $O(n)$ space, requires $\oplus$ to be invertible, but does not require the window to be FIFO.

This paper introduces the *De-Amortized Banker's Aggregator (DABA)*, a novel general-purpose sliding-window aggregation algorithm that guarantees low-latency response on every operation—in the worst case, not just on average. The algorithm is simple, and supports both fixed-sized and variable-sized windows. It works as long as (i) the aggregation operator, denoted by $\oplus$ in this paper, is an associative binary operator and (ii) the window has first-in first-out (FIFO) semantics. More precisely, DABA supports each of the `query`, `insert`, and `evict` operations by making at most a constant number of calls to the $\oplus$ operator in the worst case. This is independent of the window size, denoted by n in this paper.

Prior to this work, the fastest algorithms in the published literature required $O(\log n)$ calls to the $\oplus$ operator for each of the `query`, `insert`, and `evict` operations. These algorithms keep some number of partial sums in the form of a balanced aggregation tree or dyadic intervals [5, 20, 27, 29]. Some of these algorithms also support non-FIFO windows but at the expense of occasionally spending $O(n)$ to grow, shrink, or compact the data structure.

However, for the FIFO setting considered in this paper, we can formulate a faster algorithm by generalizing an idea posted on Stack Overflow for maintaining the minimum of a queue [2, 24]. The *Two-Stacks* algorithm in Figure 1 uses an old trick from functional programming to implement a queue with two stacks, front F and back B. Each stack element contains a value `val` and an aggregation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBS '17, Barcelona, Spain
© 2017 ACM. 978-1-4503-5065-5/17/06…$15.00
DOI: 10.1145/3093742.3093925

```
1  fun query()
2    return Σ_F^⊕ ⊕ Σ_B^⊕
3  fun Σ_F^⊕: if F.isEmpty() return 0̄ else return F.top().agg
4  fun Σ_B^⊕: if B.isEmpty() return 0̄ else return B.top().agg
5  fun insert(v)
6    B.push(v, Σ_B^⊕ ⊕ v)
7  fun evict()
8    if F.isEmpty() // Flip
9      while not B.isEmpty()
10       F.push(B.top().val, B.top().val ⊕ Σ_F^⊕)
11       B.pop()
12   F.pop()
```

Figure 1. Two-Stacks algorithm.

```
1  fun query()
2    return agg
3  fun insert(v)
4    vals.pushBack(v)
5    agg ← agg ⊕ v
6  fun evict()
7    v ← vals.popFront()
8    agg ← agg ⊕ inv(v)
```

Figure 2. Subtract-on-Evict algorithm.

```
1  fun query()
2    agg ← 0̄
3    for each v in vals
4      agg ← agg ⊕ v
5    return agg
6  fun insert(v)
7    vals.pushBack(v)
8  fun evict()
9    vals.popFront()
```

Figure 3. Recalculate-from-Scratch algorithm.

agg of everything below it on the stack. Insertions push on B and evictions pop from F. When F is empty, the algorithm flips B onto F, taking n invocations of $\oplus$. But this cost is amortized over the n insertions that pushed the elements in the first place. That means each of *query*, *insert*, and *evict* invokes the $\oplus$ operator, *on average*, a constant number of times. Unfortunately, due to the occasional $O(n)$ operation, we measured the standard deviation of the latency to be 12–54× the average latency for Two-Stacks.

In theoretical terms, Two-Stacks is *amortized* constant whereas DABA is *worst-case* constant. As far as we know, DABA is the first algorithm to guarantee a worst-case constant bound for any binary associative operator and a FIFO window. The design for DABA builds on the following two ideas:

- FIFO sliding-window aggregation can be implemented in average-case $O(1)$ time using the Two-Stacks algorithm.
- A FIFO data structure can be maintained in worst-case $O(1)$ time using, for example, Okasaki's functional queue [22].

We derive DABA by extending these ideas in several directions. In broad strokes, Okasaki's functional queue avoids occasional high-latency actions by spreading them out roughly equally over multiple operations. Though some care is required, combining these ideas results in what we call *Functional Okasaki Aggregator (FOA)*, a functional implementation that yields an $O(1)$ worst-case bound for all three operations. Implementing FOA as described requires a functional language with lazy evaluation and automatic garbage collection. By relying on simple reference counting in place of garbage collection, and manually tailoring lazy evaluation for this particular case, we derive *Imperative Okasaki Aggregator (IOA)*, which can be implemented in C++. IOA, however, is not memory-friendly. DABA takes this one step further: instead of a recursive functional data structure, it uses an efficient flat representation while being able to perform the high-latency actions gradually so each operation incurs at most four calls to the $\oplus$ operator.

Beyond FIFO windows, various other settings have been studied. For invertible operators, Subtract-on-Evict (Figure 2) delivers great performance. By keeping a running sum, it invokes $\oplus$ at most $O(1)$ times per window change. But this requires $\oplus$ to be invertible, which many operators of prime interest are not. To handle non-invertibility, one can aim for generality: Recalculate-from-Scratch (Figure 3) keeps a queue of window contents that can be walked from front to back, requiring $O(n)$ invocations to $\oplus$ per *query*. However, the linear time complexity makes it infeasible for large n. One way to side-step this issue is via coarse-grained windows where evictions occur in batches. By pre-aggregating inside each batch, the cost of aggregating across batches shrinks. But coarse-grained windows are an approximation that does not always satisfy application requirements. Handling the non-invertible case efficiently is more involved and has been extensively studied [5, 17, 19, 20, 27, 29]. Table 1 summarizes this discussion. DABA takes a large step forward by supporting non-invertible operators in worst-case constant time.

Experiments show that DABA performs well in practice. We have implemented our new algorithm in C++ and benchmarked it against several alternative approaches. The results show that DABA has small overhead: only slightly slower than naïve approaches for very small windows. For moderate and large windows, it incurs much better latencies than existing algorithms, thanks to the asymptotic differences. When the associative operator of the underlying aggregation is constant-time, then DABA offers constant-time sliding-window aggregation with a small constant.

2 Problem Definition

This section formalizes the problem of maintaining aggregation in a first-in first-out sliding window and discusses the kinds of aggregations supported in this work.

2.1 Sliding-Window Aggregation Data Type

Sliding-window aggregation usually operates 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 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 count-based window evicts the earliest data item to keep the size fixed [12]). Window eviction policies are orthogonal to the algorithms in this paper.

This section models 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 an algebraic structure called a monoid:

Definition: 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

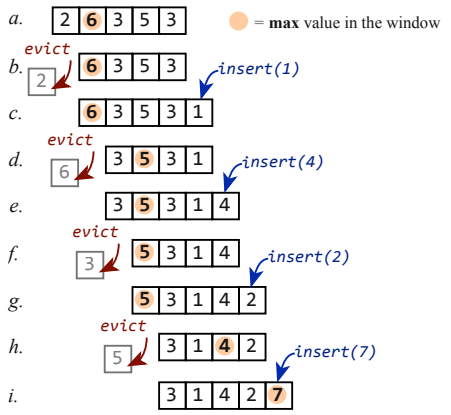


Figure 4. SWAG example trace. The sliding-window maintains the maximum value (bolded and surrounded by a shaded circle).

– *Identity:* $\bar{0} \in S$ is the identity: $\bar{0} \oplus a = a = a \oplus \bar{0}$ 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 $\bar{0}$ is a generalization of the number zero.

A monoid is *commutative* if $a \oplus b = b \oplus a$ for all $a, b \in S$. A 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 was demonstrated before [8, 27, 30]. For this reason, we focus our attention on supporting monoidal aggregation, formulating the abstract data type as follows:

Definition: The first-in first-out *sliding-window aggregation* (SWAG) abstract data type maintains a collection of window data and supports the following operations:

- `insert`(v) adds v to the rear of the sliding window. That is, if the sliding window contains values $v_0, v_1, \ldots, v_{n-1}$ 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 to say, 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` returns $v_0 \oplus v_1 \oplus \cdots \oplus v_{n-1}$. If the window is empty, it returns $\bar{0}$.

Throughout the paper, n will denote the size of the current sliding window and $v_0, v_1, \ldots, v_{n-1}$ will denote the contents of the sliding window in their arrival order, where v_0 is the oldest element.

Example: As a running example, Figure 4 shows a typical interaction with the SWAG data type. The example SWAG uses 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. Even though in this trace, `insert` and `evict` alternate, the SWAG data type, as well as all our algorithms, places no restrictions on how `insert` and `evict` may be called. They can be arbitrarily interleaved, allowing for dynamically-sized windows.

2.2 Aggregation on Monoids

Despite their simplicity, monoids are expressive enough to capture most basic aggregations [8, 27], as well as more sophisticated aggregations such as maintaining approximate membership via a Bloom filter [7], maintaining an approximate count of distinct elements [11], and maintaining the versatile count-min sketch [9].

However, many aggregations (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 [27] gives a framework for the developer to provide three types In, Agg, and Out and write three functions as follows:

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

As an example, for arithmetic mean, we write:

$$\begin{aligned} \texttt{lift}(e) &= \{n \leftarrow 1, \Sigma \leftarrow e\} \\ \texttt{combine}(v_1, v_2) &= \{n \leftarrow v_1.n + v_2.n,\ \Sigma \leftarrow v_1.\Sigma + v_2.\Sigma\} \\ \texttt{lower}(a) &= a.\Sigma / a.n. \end{aligned}$$

In this framework, a query is *conceptually* answered as follows. If the sliding window currently contains 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 $a = v_0 \oplus v_1 \oplus \cdots \oplus v_{n-1}$. 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. Therefore, the present paper focuses exclusively on the issue of maintaining the monoidal sum—i.e., how to call `combine` as rarely as possible.

3 Okasaki-Based Aggregators

Our first two aggregation algorithms build upon Okasaki's real-time queue [22]. This queue is a FIFO data structure where adding and removing an element takes worst-case $O(1)$ time. But the queue does not support aggregation. In this section, we review Okasaki's algorithm and extend it to support aggregation. Our algorithms invoke $\oplus$ at most $O(1)$ times per operation, even in the worst-case.

While this already represents theoretical improvements to the state of the art, the algorithms require laziness (a feature missing from most main-stream programming languages) and make heavy use of fine-grained memory allocation and pointer-chasing. Our experiments found that this hampers performance. The next section will address these issues (impatient readers may want to skip ahead to Section 4).

3.1 Functional Okasaki Aggregator (FOA)

We begin by describing Functional Okasaki Aggregator (FOA). The main idea behind FOA is to incrementalize the Two-Stacks algorithm. As common in functional programming, stacks are referred

to as lists. FOA uses only functional programming features and avoids mutating assignments. The basis for FOA is lazy functional lists with aggregations supporting the following operations:

- `[]` constructs an empty list.
- `cons`(v, a, ℓ) constructs a list with the first value v, aggregation a, and tail list ℓ. Most notably, it is lazy in ℓ: If a call to `cons` passes an expression to ℓ, that expression is not evaluated at the time of `cons`, but postponed until that tail list is inspected. This may happen much later or never at all. The laziness of the ℓ argument is essential for FOA's algorithmic complexity as it prevents work from happening all at once, which would break the worst-case $O(1)$ time bound. The ℓ argument to `cons` is the only place where FOA uses laziness.
- `head`(ℓ) retrieves the first value from list ℓ.
- `agg`(ℓ) retrieves the aggregation from list ℓ.
- `tail`(ℓ) retrieves the tail list from list ℓ.

FOA always uses `cons`(v, a, ℓ) such that either $a = v \oplus$ `agg`(ℓ) or $a =$ `agg`$(\ell) \oplus v$, depending on whether the list being constructed represents elements from the sliding window in forward or reverse order. At this point, it may seem natural to bake the calculation for a into `cons`. However, this would not work because literally calling `agg`(ℓ) on the tail ℓ would force its evaluation and thus break the $O(1)$ behavior. Instead, FOA is designed to derive the aggregation result out-of-band and pass that to `cons`.

The FOA data structure is a triple $\langle F, N, B\rangle$ of lists. List F (front) contains the oldest window elements, list N (next) is a sublist of F, and list B (back) contains the newest window elements. In other words, F and B correspond to the two stacks of the Two-Stacks algorithm, and N is a pointer into the interior of F. To support both fast eviction from the front and fast insertion at the back, lists F and B store their elements in the opposite order: the head of F is the oldest (first-in) element, whereas the head of B is the newest (last-in) element. Since evictions would eventually drain F, the algorithm must occasionally move elements from B to F while rotating them, i.e., reversing their order.

To operate in worst-case $O(1)$, the rotation must not happen in one fell swoop. Okasaki shows how to carefully suspend and resume rotation (`rot`) one element at a time using laziness. To accomplish this, N points to the next element to rotate, and if that element contains a suspended `rot`, then `tail`(N) implicitly forces it. Fortunately, Okasaki's idea can be naturally extended to incrementalize not just the FIFO queue but also its aggregation.

Figure 5 shows the FOA algorithm. We based FOA on Figure 4 of Okasaki's paper [22], but we extended it for aggregation and changed some variable names for clarity. Function `query` just returns the monoidal sum of F and B. Function `insert`(v) adds the new element v to B, then calls `makeq` for one step of incremental rotation. Similarly, function `evict` drops an element from F, then calls `makeq` for one step of incremental rotation. Function `makeq` has two cases. If N is empty (Line 8), that means the previous incremental rotation has finished, and it is time to start a new one. Otherwise (Line 11), `makeq` calls `tail`(N) to advance N by one element, thus implicitly forcing one suspended `rot` step if N happened to be pointing to a suspended invocation of `rot`.

Function `rot`(L, R, A) takes three lists as parameters, and returns the concatenation of L (left) with the reversal of R (right) and with A (accumulator). Its precondition is that $|R| = |L| + 1$, so it can terminate when L is empty (Line 15). Since `cons` is lazy in its last argument, the recursive call in Line 19 is not evaluated eagerly, and

```
1  fun query(⟨F,N,B⟩)
2    return agg(F) ⊕ agg(B)
3  fun insert(v, ⟨F,N,B⟩)
4    return makeq(⟨F, N, cons(v, agg(B) ⊕ v, B)⟩)
5  fun evict(⟨F,N,B⟩)
6    return makeq(⟨tail(F), N, B⟩)
7  fun makeq(⟨F,N,B⟩)
8    if isEmpty(N)
9      F' ← rot(F, B, [])
10     return ⟨F', F', []⟩
11   else
12     return ⟨F, tail(N), B⟩
13 fun rot(L, R, A)
14   A' ← cons(head(R), head(R) ⊕ agg(A), A)
15   if isEmpty(L)
16     return A'
17   else
18     return cons(head(L), agg(L) ⊕ agg(R) ⊕ agg(A),
19                 rot(tail(L), tail(R), A'))
```

Figure 5. Functional Okasaki Aggregator (FOA) algorithm.

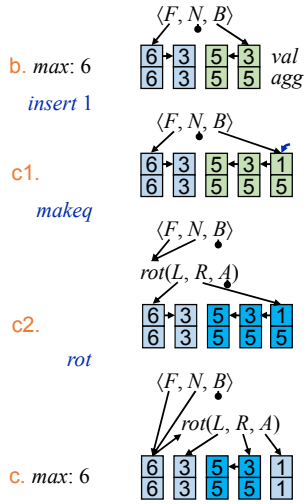


Figure 6. Functional Okasaki Aggregator (FOA) trace detail b→c.

instead results in a suspended invocation of `rot`. Figure 6 illustrates this. Step b→c1 inserts an element to B (Line 4). Step c1→c2 is a call to `makeq` with empty N. It thus executes Lines 9-10, calling `rot` and binding both F and N to the result. Step c2→c eagerly executes the first call to `rot`, but Lines 18-19 lazily call `cons`. Hence, execution quiesces in State c with a suspended recursive invocation of `rot` (Line 19) that is postponed to future sliding-window operations.

Figure 7 illustrates how FOA works on the entire running example of SWAG trace from Figure 4. We already saw the details of Step b→c, which introduces a new suspended `rot`. Pointer N advances by one element in each subsequent step until Step g→h (reaching the end of F). Along the way, N forces suspended `rots` when reaching them until Step e→f: R is empty and `rot` disappears.

We claimed earlier that FOA "knows" aggregations to pass to `cons` without needing to force suspended invocations of `rot`. To substantiate this claim, we first need a lemma:

Lemma 3.1. *FOA has at most one suspended invocation of* `rot`*. Furthermore, this suspended* `rot` *is in F, not in B.*

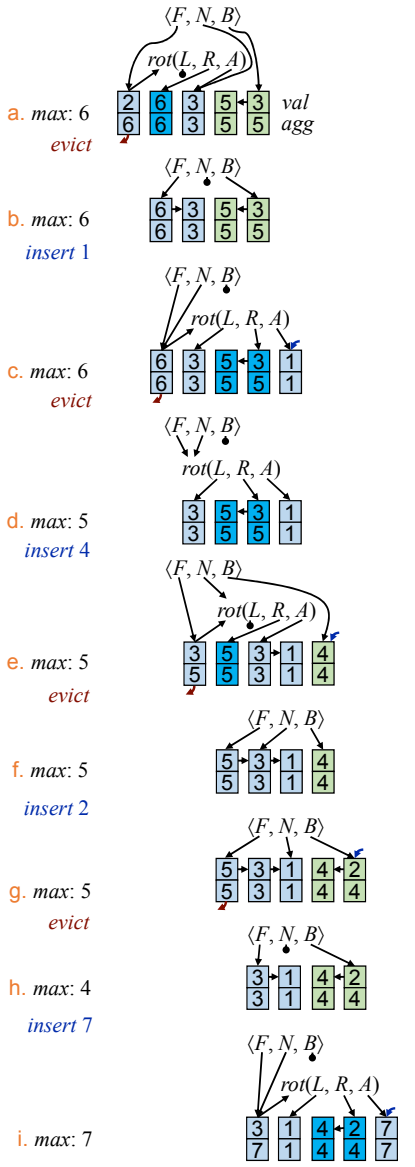


Figure 7. Functional Okasaki Aggregator (FOA) sample execution.

The proofs for Lemma 3.1, as well as for Theorems 3.2 and 3.3 below, are in Appendix A. Because of Lemma 3.1, the aggregation arguments to `cons` in Lines 4, 14, and 18 only refer to fully reified lists without suspensions. Okasaki proves the following invariant about his FIFO queue [22]:

$$|F| \leq |B| \qquad \text{and} \qquad |N| = |L| - |R|$$

This invariant ensures that the algorithm always finishes rotations in time to enable evictions from F.

Theorem 3.2. *If the window currently contains values $v_0, \ldots, v_{n-1}$, FOA* `query` *returns $v_0 \oplus \ldots \oplus v_{n-1}$.*

Theorem 3.3. *Each invocation of FOA* `query`*,* `insert`*, or* `evict` *makes at most $O(1)$ invocations of $\oplus$.*

Theorem 3.2 establishes the correctness and Theorem 3.3 establishes the algorithmic complexity of FOA.

3.2 Imperative Okasaki Aggregator (IOA)

Building on Okasaki's real-time queue, FOA requires a programming language that offers lazy evaluation and automatic garbage collection. Furthermore, strictly speaking, worst-case $O(1)$ behavior requires real-time garbage collection. While both features exist (e.g., Haskell is a language with lazy evaluation [15] and the Metronome is a real-time garbage collector for Java [6]), neither feature is mainstream. Furthermore, we wanted to experimentally determine the performance of different implementation strategies and language compilers. Therefore, we also implemented a second algorithm IOA, for Imperative Okasaki Aggregator, in C++.

For memory reclamation, IOA uses reference counting implemented by `std::shared_ptr` in the C++ standard library. For laziness, IOA uses a hand-coded C++ class that can represent a list cell as either forced or delayed. A forced list cell simply holds a value, an aggregation, and a tail pointer to another list. Since the only real use of laziness in FOA is the call to `rot`, a delayed list cell holds the parameters to the suspended `rot`, i.e., L, R, and A. Unfortunately, these implementation techniques make the code unnatural. Furthermore, IOA still uses fine-grained memory allocation and pointer chasing, imposing a performance penalty on modern architectures.

3.3 Beyond Okasaki-Based Aggregators

FOA and IOA introduced in this section are the first worst-case $O(1)$ sliding-window aggregation algorithms. However, FOA uses uncommon language facilities and IOA uses unnatural implementation techniques. Worse, Section 5 demonstrates that the performance of both algorithms is hampered by the use of one heap-allocated object per list element. In the next section, we aim to retain the worst-case $O(1)$ theoretical guarantee but reduce the overhead by avoiding a large number of small memory allocations and pointer chasing. We will exploit two observations: (1) There is at most one suspended `rot` (Lemma 3.1), so we hold a few pointers on the side instead of in fields of individual list elements. (2) We purposely visualized the execution trace in Figure 7 such that list elements do not move horizontally on the page. When a list element appears in two consecutive states, it does not move left or right. If we imagine the horizontal position to correspond to a memory location, this gives us a hint that list elements may not need to move in memory throughout the execution, avoiding spurious copying.

4 De-Amortized Banker's Aggregator (DABA)

This section introduces the De-Amortized Banker's Aggregator (DABA). DABA is a SWAG (sliding window aggregator) with worst-case $O(1)$ invocations of $\oplus$ per SWAG operation. It is inspired by the Okasaki-based algorithms from the previous section. But it does not need laziness and avoids fine-grained memory allocation, as well as spurious copies. DABA is called so because of the ingredients involved in the algorithm: Amortization looks at the average cost of an operation over a long period of time. The banker's method conceptualizes amortization as money movements between the algorithm and a fictitious bank. Deamortization is a method that turns the average-case behavior into the worst-case behavior, usually by carefully spreading out expensive operations. To aid readability, we wrote this section to be be understood independently of FOA and IOA (in case the reader skipped Section 3).

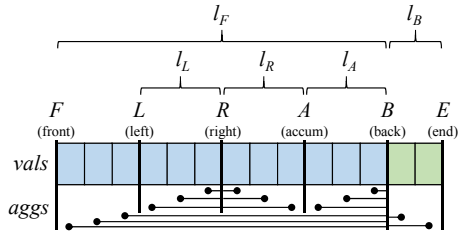
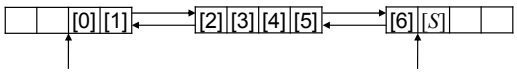


Figure 8. DABA data structure.

4.1 Chunked-Array Queue

To implement a worst-case $O(1)$ SWAG, DABA needs a worst-case $O(1)$ FIFO (first-in first-out) data structure. Furthermore, the FIFO should avoid fine-grained memory allocation. As it turns out, such a FIFO is easy to build using a doubly-linked list of chunks:



Each chunk is a fixed-size array of elements. The queue starts possibly in the middle of a chunk, continues through chunks in the order of forward links, and ends possibly in the middle of a chunk. Upward arrows in the figure are pointers to the beginning and the end of the queue. In addition, there can be pointers to other queue elements. Such queue pointers are represented by the address of a chunk and an index into the chunk. To ensure that the end pointer is well-formed even in the boundary case where the end of the queue aligns with the end of a chunk, the implementation reserves a sentinel [S] behind the last queue element. The chunked-array queue provides the following operations in worst-case $O(1)$ time:

- q.`pushBack`(v) inserts element v at the end of queue q.
- q.`popFront`() evicts the first element from queue q.
- $p_1 \leftarrow p_2$ overwrites pointer p_1 with p_2, changing p_1 to point to the same element as p_2.
- $p_1 = p_2$ compares pointers p_1 and p_2 for equality.
- $p + 1$ returns a pointer to the next element after p.
- $p - 1$ returns a pointer to the previous element before p.
- $q[p]$ reads the element of queue q at pointer p.
- $q[p] \leftarrow v$ overwrites the element of q at pointer p with v.

Notice that the interface here hides the chunking, and the implementation is imperative, not purely functional. It uses mutating assignments and in-place updates, so it can avoid spurious copies. The only memory allocation happens when `pushBack` needs to create a new chunk; a chunk is freed when it becomes unused. Allocation frequency is inversely proportional to chunk size.

Furthermore, we will state DABA's correctness invariants in terms of two additional operations (because DABA never calls them, they do not affect the complexity bounds):

- $p + i$ advances p a total of i times (calling $+1$ i times).
- $p_1 \le p_2$ returns `true` if there exists a non-negative integer i such that $p_1 + i = p_2$ and `false` otherwise.

4.2 DABA Data Structure

DABA uses a data structure consisting of two queues, *vals* and *aggs*, as shown in Figure 8. Both are implemented as chunked-array queues with several pointers F, L, R, A, B, and E. The pointers are always ordered as follows:

$$F \le L \le R \le A \le B \le E$$

```
1 fun Σ⊕_F: if (F = B) return 0̄ else return aggs[F]
2 fun Σ⊕_B: if (B = E) return 0̄ else return aggs[E - 1]
3 fun Σ⊕_L: if (L = R) return 0̄ else return aggs[L]
4 fun Σ⊕_R: if (R = A) return 0̄ else return aggs[A - 1]
5 fun Σ⊕_A: if (A = B) return 0̄ else return aggs[A]
```

Figure 9. DABA helper functions.

Queue *vals* stores the window contents, with the i^{th} oldest value in FIFO order stored at $v_i = vals[F + i]$.

Queue *aggs* stores partial aggregations over subranges of *vals*. Specifically, the •– and –• notation in Figure 8 indicates that $aggs[\bullet]$ holds the monoidal sum of the values above the horizontal line –. For example, $aggs[F]$ holds $vals[F] \oplus \ldots \oplus vals[B-1]$. Each pointer p corresponds to a sublist l_p as shown in Figure 8. For example, pointer F corresponds to sublist l_F. Each list is either aggregated to the left •– or to the right –•. The direction is carefully chosen to enable the SWAG operations. The front list l_F is aggregated to the left to facilitate eviction. The back list l_B is aggregated to the right to facilitate insertion. The sub-lists l_L, l_R, and l_A in the middle are designed to facilitate incremental reversal. Incremental reversal happens by adjusting the pointers demarcating sublist boundaries one step at a time. When a pointer moves, an element of *aggs* changes membership from one sublist to another and may need to be updated accordingly.

DABA's *aggs invariants* specify the contents of *aggs* before and after each SWAG operation. The •– and –• notation in Figure 8 is a graphical depiction of the *aggs* invariants. Formally, the *aggs* invariants are:

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

Figure 9 defines helper functions for returning the aggregation of each of the sub-lists. Each helper function returns the identity element $\bar{0}$ if the corresponding sublist is empty or otherwise reads the aggregation from the appropriate element of *aggs*. As Section 4.4 will show, if the window is non-empty, then there is at least one element between F and L, so $\Sigma^{\oplus}_F$ returns the correct value.

4.3 DABA Algorithm

Figure 10 shows the DABA algorithm. Function `query` just returns the monoidal sum of the front and back lists. Function `insert` pushes a new value and the corresponding aggregation on the back (l_B) of *vals* and *aggs*, then calls `fixup`. Similarly, function `evict` pops one value and the corresponding aggregation from the front (l_F) of *vals* and *aggs*, then calls `fixup`. Function `fixup` performs the incremental reversal of l_B into l_F. This is the most subtle part of the DABA algorithm. Before diving into the details of the code, we will discuss `fixup` at a higher level using an example.

Figure 11 illustrates how DABA works on the running example trace from Figure 4. Each state shows the contents of the *vals* and *aggs* queues along with the pointers. For instance, in State a, the entire window is in the front list l_F (shown in light blue), and the back list l_B is empty ($B = E$; other states show l_B in light green). In State a, the pointers L, R, and A differ. That means each of the corresponding sublists l_L, l_R, and l_A are non-empty, and thus, have

```
1  fun query()
2    return Σ⊕_F ⊕ Σ⊕_B
3  fun insert(v)
4    vals.pushBack(v), aggs.pushBack(Σ⊕_B ⊕ v)
5    fixup()
6  fun evict()
7    vals.popFront(), aggs.popFront()
8    fixup()
9  fun fixup()
10   if F = B      // Singleton case, Figure 12(a)
11     B ← E, A ← E, R ← E, L ← E
12   else
13     if L = B    // Flip, Figure 12(b)
14       L ← F, A ← E, B ← E
15     if L = R    // Shift, Figure 12(c)
16       A ← A + 1, R ← R + 1, L ← L + 1
17     else        // Shrink, Figure 12(d)
18       aggs[L] ← Σ⊕_L ⊕ Σ⊕_R ⊕ Σ⊕_A
19       L ← L + 1
20       aggs[A - 1] ← vals[A - 1] ⊕ Σ⊕_A
21       A ← A - 1
```

Figure 10. DABA algorithm.

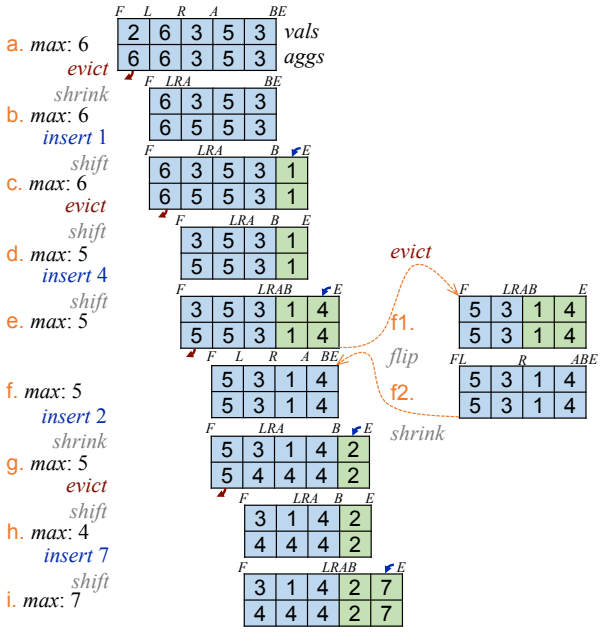
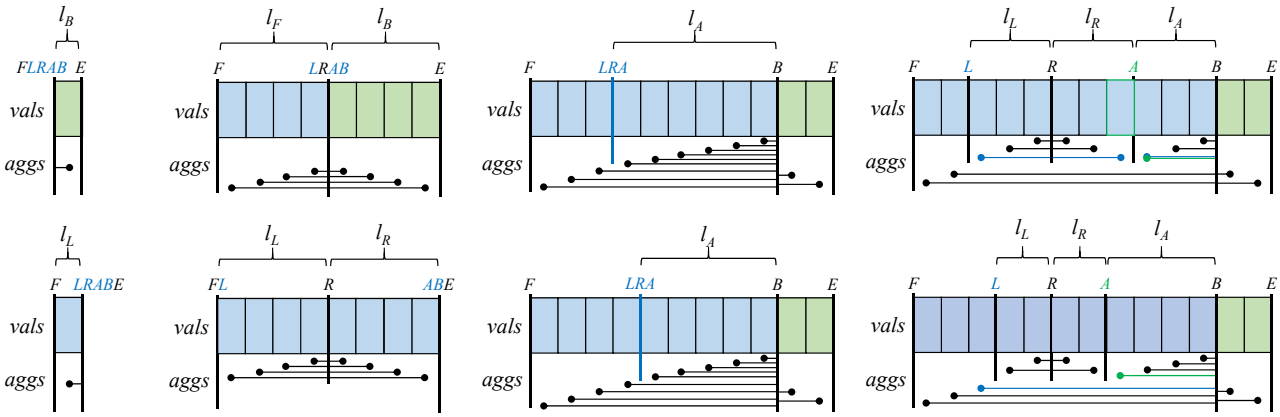


Figure 11. DABA example execution trace.



(a) Singleton case, Lines 10-11 (b) Flip, Lines 13-14 (c) Shift, Lines 15-16 (d) Shrink, Lines 17-21

Figure 12. Cases of DABA's *fixup*. The top/bottom of each case show the state before/after executing the referenced lines from Figure 10.

their own internal partial aggregation. The direction of the partial aggregations is as shown in Figure 8.

Each step from one state to the next is annotated with the SWAG operation and one case of *fixup*. For instance, Step a→b uses the *evict* operation and triggers the *shrink* case of *fixup*, shrinking l_L and l_R by one element each. Step b→c uses *insert* and triggers the *shift* case of *fixup*, incrementing pointers L, R, and A. The next two steps also trigger *shift*, until the pointers L, R, and A reach B. Next, Step e→f triggers the *flip* case, relabeling the old l_F and l_B into the new l_L and l_R. Next come two more *shrink*s and two more *shift*s. If the trace were to continue, the next step would *flip* again.

Generalizing from this concrete example, we discuss the code of *fixup* in detail. If the window has at least two elements, in the long run, the *fixup* cases form the pattern $(shrink^+\ shift^+\ flip)^*$. Figure 12 shows four cases of *fixup*:

***Singleton* case:** Figure 12(a). If $F = B$, that means the front list l_F is empty. As Section 4.4 will show, that can only happen if the back list l_B has exactly one element. On a singleton list, there is no difference between aggregating to the left or right. Therefore, DABA can simply move pointers around to turn l_B into l_F without having to modify *aggs*.

***Flip* case:** Figure 12(b). If $F \neq B$ but $L = B$, that means that the three sublists l_L, l_R, and l_A of l_F are all empty. In that case, the entire l_F is aggregated to the left •– and l_B is aggregated to the right –•. So DABA can simply move pointers around to turn l_F and l_B into l_L and l_R, which are aggregated in the same directions.

***Shift* case:** Figure 12(c). If $L \neq B$ but $L = R$, that means that l_L and l_R are empty but l_A is non-empty. That means that all of l_L is aggregated to the left •–. In other words, the boundary of l_A makes no difference for the aggregation, and DABA can increment L, R, and A without having to modify *aggs*.

***Shrink* case:** Figure 12(d). If $L \neq R$, that means that l_L is non-empty. As Section 4.4 will show, l_L and l_R always have the same length, so l_R is non-empty too. DABA shrinks both l_L and l_R by one element each. This is the only case that requires modifying *aggs*. DABA shrinks l_L by incrementing L, thus moving the first element of l_L to the front portion of l_L; the new *aggs* entry for that element is $\Sigma_L^{\oplus} \oplus \Sigma_R^{\oplus} \oplus \Sigma_A^{\oplus}$. DABA shrinks l_R by decrementing A, thus moving the last element of l_R to l_A; the new *aggs* entry for that element is $vals[A-1] \oplus \Sigma_A^{\oplus}$.

We have now described the DABA algorithm; we establish correctness and running time next.

4.4 DABA Theorems

This section states the theorems that DABA is correct and is worst-case $O(1)$. The theorems depend upon DABA's *size invariants*, which specify the sizes of sublists. The size invariants state that the window is either empty ($|l_F| = 0$ and $|l_B| = 0$) or the following two conditions hold:

- *First*, $|l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B|$. The left side of this equation is the number of incremental reversal steps required to shrink and shift the sublists of l_F until they are empty, plus one element in the front portion of l_F to offer easy access to $\Sigma_F^{\oplus}$. The right side is the number of available steps until the next reversal must start, because waiting longer would build up too much reversal work to handle incrementally.
- *Second*, $|l_L| = |l_R|$. After each *flip*, l_L and l_R start out with the same size and then shrink at the same pace.

Lemma 4.1. *DABA maintains the following size invariants:*

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

Theorem 4.2. *If the window currently contains values $v_0, \ldots, v_{n-1}$, DABA* `query` *returns $v_0 \oplus \ldots \oplus v_{n-1}$.*

Theorem 4.3. *DABA invokes $\oplus$ at most one time per* `query`*, four times per* `insert`*, and three times per* `evict`*. Furthermore, for non-empty windows, DABA invokes $\oplus$ on average 2.5 times per* `insert` *and 1.5 times per* `evict`*.*

Theorem 4.2 establishes the correctness, proved in Appendix A. It follows from Theorem 4.3 that each invocation of DABA *query*, *insert*, or *evict* makes at most $O(1)$ invocations of $\oplus$. A simple caching optimization can reduce the number of $\oplus$ invocations even further beyond Theorem 4.3 [26].

We further remark that DABA supports variable-sized windows since all theorems in this section 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 operator $\oplus$ is associative but does not need to be commutative or invertible.

5 Experimental Evaluation

Our experimental evaluation has two main purposes: to test whether the worst-case constant behavior of FOA, IOA, and DABA translates into consistently low latency and high throughput in practice; and to determine when the aggregation algorithms are faster than recalculating an aggregation over a window from scratch.

Operator	Average ± standard deviation of latency in cycles			
	Two-Stacks	FOA	IOA	DABA
Sum	127 ± 2,980	250 ± 15,559	1,824 ± 1,919	193 ± 113
Max	125 ± 2,991	332 ± 15,356	1,831 ± 2,083	200 ± 122
ArgMax	136 ± 3,214	378 ± 18,693	1,846 ± 1,925	207 ± 137
MinCount	154 ± 4,463	395 ± 20,150	1,937 ± 2,103	222 ± 157
ArithMean	165 ± 3,740	397 ± 18,528	1,898 ± 2,084	237 ± 149
StdDev	200 ± 4,099	469 ± 20,373	1,943 ± 2,081	274 ± 160
GeoMean	299 ± 3,777	484 ± 18,171	2,055 ± 2,100	366 ± 168
Bloom	5,532±298,982	12,394±310,698	21,595±14,691	9,021±4,249

Table 2. Lantencies in processor cycles, from a run with a 2^{14} data item window over 1 million rounds of `insert`, `evict`, `query`.

Our experiments use up to six different SWAGs: Two-Stacks, FOA, IOA, DABA, an implementation of the Reactive Aggregator [27] (*Reactive*), and recalculating the window from scratch (*Recalc*). Two-Stacks is amortized constant, while FOA, IOA, and DABA are worst-case constant. Reactive serves as our comparison against the current state of the art; all operations on it are amortized $O(\log n)$. Recalc is our performance baseline. We implemented FOA in Standard ML by modifying Okasaki's own implementation of his functional real-time queue, and compiled it using the MLton compiler cloned from GitHub on June 24, 2016. We implemented all of the other algorithms, including IOA, in C++11, using the g++ compiler version 4.8.3 with optimization level *-O3*. Our goal was to make the comparison to Okasaki's algorithms as fair as possible, which is why we compare against both FOA and IOA.

Our benchmark driver starts with a ramp-up to grow the window size to n. Then, it starts measuring performance for a number of iterations. Each iteration of the driver issues an `evict`, `insert`, and `query` to the SWAG. Some of the algorithms we evaluate take as little as 100-200 processor cycles per iteration. Because of this, we conduct all of our experiments outside of existing streaming platforms to better isolate and measure performance effects. All of the algorithms can be easily integrated into existing streaming platforms (see for instance Reactive [27]).

We chose a representative sample of aggregation operators for our experiments. They range from operators that take a single instruction (Sum, Max), to operators that take many thousands of instructions (Bloom), to inherently linear operators (Collect). Our experimental system runs RedHat 7.2, with Linux kernel version 3.10.0. The processor is an Intel Xeon E5-2680 at 2.7 GHz.

5.1 Latency

The practical benefit of using an aggregation algorithm with theoretical worst-case constant complexity should be tight latency bounds. For an amortized constant algorithm, such as Two-Stacks, we expect to see a low average latency, but a high standard deviation due to periodic high-cost operations. However, for the worst-case constant algorithms, we expect to see a low average latency and a low standard deviation. Table 2 shows that our expectations hold for Two-Stacks and DABA, but not for FOA and IOA. These experiments used a window size of 2^{14} data items over 1 million rounds of `evict`, `insert`, and `query` over our set of aggregation operators.

Two-Stacks has the lowest average latencies across all aggregation operators, but also has periodic latency spikes, resulting in a standard deviation that is between 12–54× the average. DABA maintains an average latency that is about 20–60% higher than

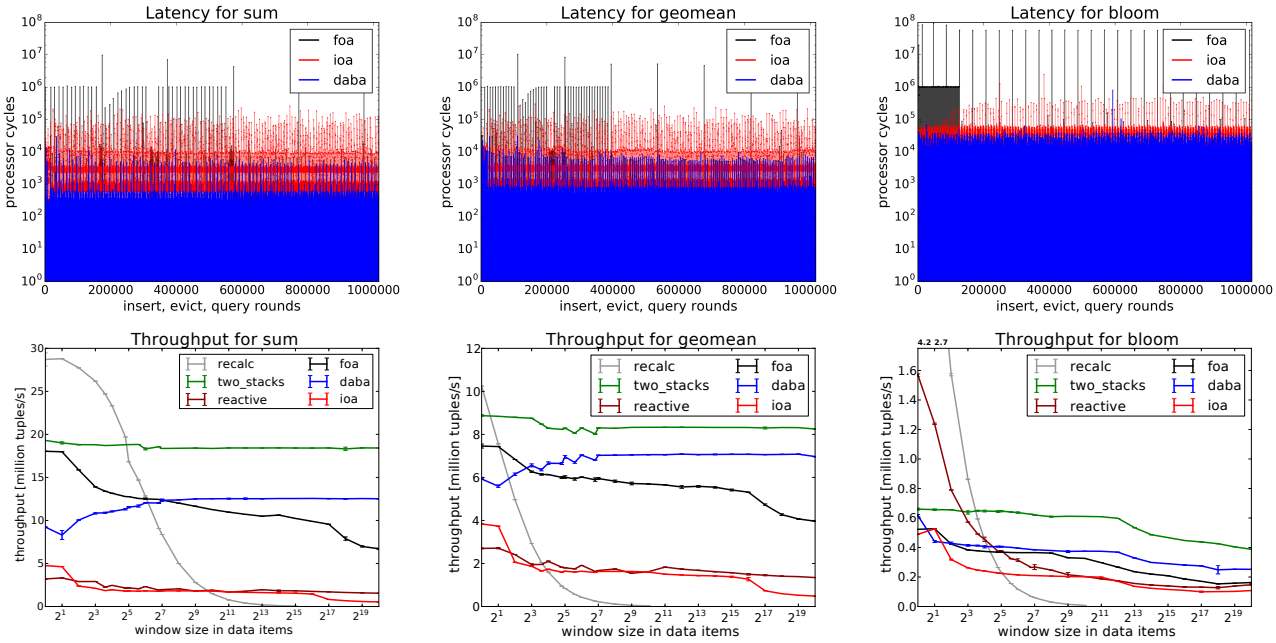


Figure 13. Top: Latency experiments. Each graph is a single run with a window of 2^{14} data items. The x-axis is the number of rounds of *evict*, *insert*, *query* into the experiment, up to 1 million rounds. The y-axis is the number of processor cycles to execute that round on a log scale. **Bottom:** Throughput experiments. The x-axis is the window size in number of data items in the SWAG, and the y-axis is throughput in million data items per second.

Two-Stacks, but its standard deviation is always lower than the average itself.

FOA's average latencies are about 1.6–2.8× as expensive as Two-Stacks, but the standard deviations are as much as 5× higher. FOA, a worst-case constant algorithm, has latencies significantly higher than an amortized constant algorithm. That is, even though Two-Stacks has periodic linear costs, FOA still performs worse. We attribute this discrepancy to the cost of garbage collection. Our algorithmic analysis looks at the number of invocations of ⊕, but does not consider the cost of memory management. In the case of FOA, this memory management is significant enough that it is never profitable to use FOA over an amortized constant algorithm. IOA also has a high cost due to memory management, which again is not reflected in the theoretical analysis: its average latencies are 6–14× those of Two-Stacks. On the other hand, IOA's standard deviation is lower than Two-Stacks', usually within 2×. The only exception is Bloom: IOA's standard deviation is about 20× lower.

The top of Figure 13 details the latency behavior for the three worst-case constant algorithms, FOA, IOA, and DABA. It shows a latency time-series for a single run with a window size of 2^{14} data items for three aggregation operators. We chose Sum, GeoMean, and Bloom to represent cheap, medium, and expensive operators. The x-axis is the number of *evict*, *insert*, and *query* rounds into the experiment, and the y-axis is the cost in processor cycles to execute that round on a log scale.

All three graphs show the same trends and corroborate the averages and standard deviations in Table 2. DABA's typical cost is about 10× less than IOA, and its worst cost is usually lower than IOA's best. FOA's typical cost is in the same order of magnitude as DABA, but it periodically has a cost that is around 40× more expensive. FOA also shows higher average latency near the beginning; this is caused by the garbage collector ramping up the heap size.

5.2 Throughput

Our throughput experiments, bottom of Figure 13, explore a wide range of window sizes for the same three aggregation operators. In all three experiments, we stopped collecting data for Recalc after a certain window size; it clearly has $O(n)$ behavior, and its throughput correspondingly trends towards 0.

The general trend is that Two-Stacks maintains the best throughput, DABA next, and FOA third. While DABA and FOA are both worst-case constant, the performance difference comes from FOA spending much more time on garbage collection. Reactive, which is the worst asymptotically after Recalc, is consistently outperformed by Two-Stacks, DABA and FOA. Surprisingly, Reactive and IOA show similar trends, despite Reactive being logarithmic and IOA being worst-case constant.

For Sum and GeoMean, DABA has the curious behavior that its 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 ⊕ is compared to simple pointer manipulation. For Sum and GeoMean, the manipulation of the list pointers is comparable to the cost of ⊕. For Bloom, ⊕ is substantially more expensive than manipulating the list pointers, so this effect is not noticeable.

Two-Stacks and DABA maintain constant performance at all window sizes for Sum, but show a slight degradation as the window sizes approach 2^{20} for GeoMean, and even earlier at 2^{12} for Bloom. This drop in performance is explained by a larger memory footprint causing more cache and TLB misses; Sum maintains one 32-bit integer per element, GeoMean maintains a 64-bit float and a 32-bit integer, and Bloom maintains a bitset of 16,384 bits. 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.

However, this effect is still more pronounced in FOA and IOA. Both see a significant drop in performance after 2^{16} for Sum and GeoMean. Up to this point, however, FOA does not maintain constant performance, and IOA's absolute performance is similar to that of a logarithmic algorithm. Both facts are caused by the increased memory management to support laziness and garbage collection. FOA fares significantly better as it relies on MLton's optimized garbage collector. IOA's performance for Sum and GeoMean is dominated by memory management—every insert allocates an object, and every evict deallocates an object. These memory operations are significantly more expensive than the aggregation operators themselves. For Bloom, the cost of $\oplus$ is comparable to the memory management cost, and the fact that IOA invokes $\oplus$ a constant number of times is finally noticeable. However, Reactive, which is $O(\log n)$, still has either the same or higher throughput than IOA, even with Bloom. The lesson here is simple: $O(1)$ aggregations is not enough for scalable performance in practice. Efficient underlying data structures such as those in DABA are also required.

5.3 Break-Even Points

Table 3 answers the question, *at what window size do the SWAGs become profitable compared to Recalc?* Two-Stacks is amortized constant, and FOA, IOA, and DABA are worst-case constant, but in practice, the actual constants matter.

Our results show that the break-even points of Two-Stacks, FOA, and DABA are low enough to use over Recalc for almost all window sizes. The break-even points for IOA, however, are similar to Reactive even though IOA is constant and Reactive is logarithmic. The break-even experiments look at *average* execution time. Two-Stacks, with amortized $O(1)$ time, outperforms all others.

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 those of Two-Stacks, FOA, and DABA. The one exception is Bloom, a much more expensive operator. We note that the break-even points for Reactive are higher than reported in prior work [27]. The implementations are different; in particular, our baseline in this paper, Recalc, uses the most efficient recalculation methods we could conceive, while the baseline in the prior paper was what shipped in IBM Streams.

The one exception to these trends 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 Reactive, Two-Stacks, FOA, IOA, 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 for Collect, with Recalc having the lowest overheads in total.

Operator	Window size break-even point				
	Reactive	Two-Stacks	FOA	IOA	DABA
Sum	1,024	32	128	1,024	112
Max	1,024	28	112	1,024	64
ArgMax	512	28	112	512	64
MinCount	512	28	64	1,024	48
ArithMean	1,024	32	128	1,024	112
StdDev	1,024	32	112	1,024	64
GeoMean	16	2	4	16	4
Bloom	16	12	28	48	28
Collect	never	never	never	never	never

Table 3. 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 operator, as compared to recalculating the entire window from scratch.

5.4 Result Summary

DABA's theoretic worst-case $O(1)$ cost is a significant improvement over Reactive's amortized $O(\log n)$ behavior. Our results show that this improvement holds up in practice for both latency and throughput. Our results also show that while DABA has the same theoretic worst-case $O(1)$ cost as FOA and IOA, it is both faster and has smaller latency spikes in practice. This difference in actual performance is caused by FOA and IOA relying on garbage collection and reference counting. Overall, DABA is best for latency, and Two-Stacks is best for throughput, with DABA a close second.

6 Related Work

As before, let n be the number of elements in the window and $\oplus$ the operator that combines two partial aggregation results.

In the academic literature, the fastest sliding-window aggregation algorithms use *balanced trees* [5, 20, 27, 29]. The leaves hold input values and the parent nodes combine the aggregates of their children. When the tree is balanced, it can support the SWAG operations 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 our algorithms, state-of-the-art sliding window aggregation via balanced trees supports non-invertible and non-commutative $\oplus$ operators. They use binary trees, leading to a roughly 2× space overhead, same as in our algorithms. However, our algorithms improve over the time complexity of balanced trees, from average-case logarithmic to worst-case constant.

While the best SWAG algorithms published in papers use logarithmic time, the ideas for an amortized constant-time approach appeared on Stack Overflow. Skeet proposes using a parallel stack of minima to implement a stack supporting `push`, `pop`, and `min` in $O(1)$ time [24]. However, a stack is LIFO, but sliding window aggregation is FIFO. To get a FIFO queue supporting `insert`, `evict`, and `min` in $O(1)$ time, adamax uses two stacks to implement a queue [2]. This $O(1)$ time complexity holds in the average case, but whenever the front stack is empty, `evict` requires an $O(n)$ reversal of the back stack. While adamax posted the idea for the Two-Stacks algorithm, we implemented it generally for any monoid (adamax discussed `min` only; in the case of a non-commutative monoid, one must be careful about aggregation order); and we carefully evaluated its performance. Unlike Two-Stacks, our algorithms in this paper are not just amortized $O(1)$, but worst-case $O(1)$.

Real-time queues are FIFO data structures that support `insert` and `evict` in worst-case $O(1)$ time but lack aggregation. Hood and Melville present a real-time queue using LISP lists [14], which are mutable LIFO data structures with per-element objects and pointers. They use two lists to implement a queue, and for worst-case $O(1)$ time, they trigger reversal early, keeping a duplicate copy of the front list while reversal is in progress. Okasaki presents a real-time queue that uses immutable lists with per-element objects and pointers, and in addition, needs support for lazy evaluation [22]. Like Hood and Melville, Okasaki uses two lists and reverses early, but his queue is persistent and does not keep a duplicate copy of the front list. Our FOA algorithm augments Okasaki's queue with aggregation. We show that aggregation is possible because when constructing a lazy list, the corresponding aggregation is eagerly known without having to force any part of a lazy list. Our IOA algorithm is like FOA, but needs no language-level support for lazy evaluation. And our DABA algorithm uses similar basic ideas, but is re-invented from the ground up to avoid lazy evaluation and per-element objects and pointers. Furthermore, DABA is easy to implement in mainstream languages like C++ or Java. All three of our algorithms use worst-case $O(1)$ time like real-time queues, but unlike real-time queues, also support aggregation.

Coarse-grained windows reduce the space required for sliding window aggregation [17, 19]. In the past, these space savings also improved the time complexity. For example, reducing the window size from n elements to b batches reduces the time complexity of balanced tree algorithms from $O(\log n)$ to $O(\log b)$. However, coarse-grained windows make no difference to the time complexity of our algorithms in this paper: being worst-case constant time makes the window size n irrelevant. That said, the algorithms in this paper are easy to combine with coarse-grained windows, which helps if space savings are important in their own right.

Shared sliding window aggregation optimizes for the case where many windows over the same stream are being aggregated simultaneously. The B-Int algorithm shares windows of different sizes, answering queries in $O(\log n)$ time [5]. The paired-windows algorithm shares windows of different sizes and granularities, answering queries in $O(b)$ time, where b is the least common number of batches [17]. DABA can share its `vals` queue for different window sizes and for different aggregation operations. However, it is an open problem whether more aggressive sharing can be done in better than the $O(\log n)$ time complexity of the B-Int algorithm [5].

Non-FIFO sliding window aggregation tackles the case where the nominal timestamp of stream data items is inconsistent with their actual arrival time. Srivastava and Widom propose handling this with a holding buffer for ordering input elements before they enter the window [25]. This approach is designed for use-cases where disorder is mostly small and where higher latencies are acceptable. DABA can easily be used after such a holding buffer. Krishnamurthy et al. propose pre-aggregating each data source separately and consolidating partial aggregation results as late as possible when doing an actual query [16]. This approach is designed for use-cases where data sources are internally ordered but can have large skew against each other. DABA can easily be used on the individual ordered data streams before consolidation.

Parallelism is essential for obtaining high performance in modern streaming systems [1, 3, 18, 23, 28, 31]. Using key-based partitioning, multiple replicas of expensive operators can run in parallel. Sliding-window aggregation is often done with partitioning, in which case it is easy to parallelize. On the other hand, our DABA algorithm makes sliding-window aggregation so fast that other operators in the stream graph are more likely to become the bottleneck. Running DABA on fewer cores and hosts frees up those resources for other, more expensive, parts of the stream graph.

7 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. 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 $\oplus$ in the worst case for each `insert`, `evict`, or `query` invocation, irrespective of the current window size. 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 DABA performs well compared to other sliding-window aggregation algorithms.

References

[1] 2016. Apache Flink: Scalable Batch and Stream Data Processing. https://flink.apache.org. (2016). Retrieved Aug. 2016.

[2] adamax. 2011. Re: Implement a queue in which push_rear(), pop_front() and get_min() are all constant time operations. http://stackoverflow.com/questions/4802038/. (2011). Retrieved Aug., 2016.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Conference on Very Large Data Bases (VLDB) Industrial Track*. 734–746.

[4] Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. 2011. The extensibility framework in Microsoft StreamInsight. In *International Conference on Data Engineering (ICDE)*. 1242–1253.

[5] Arvind Arasu and Jennifer Widom. 2004. Resource sharing in continuous sliding window aggregates. In *Conference on Very Large Data Bases (VLDB)*. 336–347.

[6] David Bacon, Perry Cheng, and V. T. Rajan. 2003. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *Principles of Programming Languages (POPL)*. 285–298.

[7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)* 13, 7 (1970), 422–426.

[8] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. In *Conference on Very Large Data Bases (VLDB)*. 1441–1451.

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

[10] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *International Conference on Management of Data (SIGMOD) Industrial Track*. 647–651.

[11] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms (AofA)*. 127–146.

[12] Buğra Gedik. 2013. Generic windowing support for extensible stream processing systems. *Software Practice and Experience (SP&E)* (2013), 1105–1128.

[13] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. 2013. IBM Streams Processing Language: Analyzing Big Data in Motion. *IBM Journal of Research and Development* 57, 3/4 (2013).

[14] Robert Hood and Robert Melville. 1981. Real-Time Queue Operation in Pure LISP. *Inform. Process. Lett.* 13, 2 (1981), 50–54.

[15] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. 1992. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices* 27, 5 (1992), R1–R164.

[16] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. 2010. Continuous Analytics over Discontinuous Streams. In *International Conference on Management of Data (SIGMOD)*. 1081–1092.

[17] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *International Conference on Management of Data (SIGMOD)*. 623–634.

[18] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *International Conference on Management of Data (SIGMOD)*. 239–250.

[19] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record* 34, 1 (2005), 39–44.

[20] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. 2000. Scalable Algorithms for Large Temporal Aggregation. In *International Conference on Data Engineering (ICDE)*. 145–154.

[21] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Symposium on Operating Systems Principles (SOSP)*.

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

[23] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Transactions on Computers (TC)* 64, 2 (2015), 504–517.

[24] Jon Skeet. 2009. Re: design a stack such that getMinimum() should be O(1). http://stackoverflow.com/questions/685060/. (2009). Retrieved Aug., 2016.

[25] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *Principles of Database Systems (PODS)*. 263–274.

[26] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2015. *Constant-Time Sliding Window Aggregation.* Technical Report RC25574. IBM Research.

[27] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. In *Conference on Very Large Data Bases (VLDB)*. 702–713.

[28] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm @Twitter. In *International Conference on Management of Data (SIGMOD)*. 147–156.

[29] Jun Yang and Jennifer Widom. 2001. Incremental computation and maintenance of temporal aggregates. In *International Conference on Data Engineering (ICDE)*. 51–60.

[30] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*. 247–260.

[31] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*. 423–438.

A Proofs

Proof of Lemma 3.1. The only lazy argument is the ℓ argument to `cons`. Line 19 of Figure 5 is the only case where that argument is an expression. However, Line 19 can only be reached by calling `rot` (in either Line 9 or Line 19). The algorithm only reaches Line 9 when N is empty, and thus, any previous suspended invocation of `rot` have been forced, and Line 10 puts the new `rot` in list F. A `rot` produced from Line 19 requires consuming a previous `rot` call in the same list, i.e., also F. □

Proof of Lemma 4.1. Using Hoare logic, the start of `insert` follows:

```
3  fun insert(v)
     {|l_F| = 0 ∧ |l_B| = 0 ∨ |l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| = |l_R|}
4    vals.pushBack(v), aggs.pushBack(Σ⊕_B ⊕ v)
     {|l_F| = 0 ∧ |l_B| = 1 ∨ |l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = |l_R|}
5    fixup()
```

Since `evict` can only be called when the window is non-empty, using Hoare logic, the start of `evict` behaves as follows:

```
6  fun evict()
     {|l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| = |l_R|}
7    vals.popFront(), aggs.popFront()
     {|l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = |l_R|}
8    fixup()
```

That means that when either `insert` or `evict` calls `fixup`, the following precondition holds:

$$|l_F| = 0 \wedge |l_B| = 1 \vee |l_L| + |l_R| + |l_A| = |l_F| - |l_B| \wedge |l_L| = |l_R|$$

What remains to be shown is that `fixup` fixes up the size invariants again. This is shown in the Hoare logic proof below. Numbered lines are from Figure 10; unnumbered are assertions.

```
9  fun fixup()
     {|l_F| = 0 ∧ |l_B| = 1 ∨ |l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = |l_R|}
10   if F = B      // Singleton case, Figure 12(a)
       {|l_F| = 0 ∧ |l_B| = 1}
11     B ← E, A ← E, R ← E, L ← E
       {|l_L| + |l_R| + |l_A| + 1 = |l_F| ∧ |l_B| = 0 ∧ |l_L| = 0 ∧ |l_R| = 0}
12   else
       {|l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = |l_R|}
13     if L = B   // Flip, Figure 12(b)
         {|l_L| + |l_R| + |l_A| = 0 ∧ |l_F| = |l_B| ∧ |l_L| = 0 ∧ |l_R| = 0}
14       L ← F, A ← E, B ← E
         {|l_L| + |l_R| + |l_A| = |l_F| ∧ |l_B| = 0 ∧ |l_L| = |l_R|}
15     if L = R   // Shift, Figure 12(c)
         {|l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = 0 ∧ |l_R| = 0 ∧ |l_A| > 0}
16       A ← A + 1, R ← R + 1, L ← L + 1
         {|l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| = 0 ∧ |l_R| = 0}
17     else          // Shrink, Figure 12(d)
         {|l_L| + |l_R| + |l_A| = |l_F| − |l_B| ∧ |l_L| = |l_R| ∧ |l_L| > 0}
18       aggs[L] ← Σ⊕_L ⊕ Σ⊕_R ⊕ Σ⊕_A
19       L ← L + 1
         {|l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| + 1 = |l_R|}
20       aggs[A - 1] ← vals[A - 1] ⊕ Σ⊕_A
21       A ← A - 1
         {|l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| = |l_R|}
     {|l_F| = 0 ∧ |l_B| = 0 ∨ |l_L| + |l_R| + |l_A| + 1 = |l_F| − |l_B| ∧ |l_L| = |l_R|}
```

□

Proof of Theorem 4.2. To show `query` returns $v_0 \oplus \ldots \oplus v_{n-1}$, we show that $\Sigma_F^{\oplus}$ returns the sum of l_F and $\Sigma_B^{\oplus}$ returns the sum of l_B. The function $\Sigma_F^{\oplus}$ returns $vals[F] \oplus \ldots \oplus vals[B-1]$ because of the *aggs* invariants and because if l_F is non-empty, the front portion of l_F has at least one element (because if l_F is non-empty then $|l_F| \geq |l_F| - |l_B| = |l_L| + |l_R| + |l_A| + 1$, see Lemma 4.1). The function $\Sigma_B^{\oplus}$ returns $vals[B] \oplus \ldots \oplus vals[E-1]$ because of the *aggs* invariants. Now the *aggs* invariants hold because both `insert` and `fixup` correctly maintain the *aggs* queue. In particular, `fixup` uses the cases in Figure 12, heeding the size invariants (Lemma 4.1). □

Proof of Theorem 4.3. The algorithm has no loops nor recursion. The worst-case numbers can be seen directly from the invocations of $\oplus$ in Fig 10. For 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 `evict` operation triggers a *shrink*, shrinking l_R by one and invokes $\oplus$ thrice. 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 triggers a *shift* that shrinks l_A by one without invoking $\oplus$. The next *flip* happens when l_A is empty. Hence, there is an equal number of *shrink* and *shift* cases, so that `fixup` invokes $\oplus$ three times half of the time and does not invoke $\oplus$ half of the time. This averages out to 1.5 $\oplus$-invocation per `fixup`, and thus, 2.5 $\oplus$-invocation per `insert` and 1.5 $\oplus$-invocation per `evict`. □