Optimal and General Out-of-Order Sliding-Window Aggregation

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 derives a user-defined summary of the most-recent portion of a data stream. For in-order streams, each window change can be handled in $O(1)$ time even when the aggregation operator is not invertible. But streaming data often arrive inherently out-of-order, e.g., due to clock drifts and communication delays. For such streams, prior work resorted to latency-prone buffering or spent $O(\log n)$ time for every window change, where n is the instantaneous window size. This paper presents FiBA, a novel real-time sliding window aggregation algorithm that optimally handles streams of varying degrees of out-of-orderness. FiBA is as general as the state-of-the-art and supports variable-sized windows. An insert or evict takes amortized $O(\log d)$ time, where d is the distance of the change to the window's boundary. This means $O(1)$ time for in-order arrivals and nearly $O(1)$ time for slightly out-of-order arrivals, tending to $O(\log n)$ time for the most severely out-of-order arrivals. We also prove a matching lower bound, showing optimality. At its heart, the algorithm combines and extends finger searching, lazy rebalancing, and position-aware partial aggregates. Further, FiBA can answer range queries that aggregate subwindows for window sharing. Finally, our experiments show that FiBA performs well in practice and conforms to the theoretical findings, with significantly higher throughput than $O(\log n)$ algorithms.

PVLDB Reference Format:
Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Optimal and General Out-of-Order Sliding-Window Aggregation. *PVLDB*, 12(10): 1167–1180, 2019.
DOI: https://doi.org/10.14778/3339490.3339499

1. INTRODUCTION

Sliding-window aggregation is indispensable in streaming applications that involve continuously summarizing part of a data stream. With stream processing now in widespread production in various domains, sliding-window aggregation frameworks are expected to deliver high throughput and low latency while allowing for simple expression of a wide variety of aggregation operations.

Aggregation frameworks, when feasible, apply aggregation operations incrementally—that is, by keeping running partial aggregates and modifying them in response to values arriving or leaving the window. To aid simple expression of aggregation operations, past work [12, 15, 33] proposes casting them as binary operators. Their algebraic properties (see Table 1) dictate how efficiently they can be incrementalized. While an invertible and commutative binary operator (one with an explicit "subtract" operation, e.g., sum) is the simplest to efficiently handle, associative binary operators encompass a wider class of operations (e.g., min/max, min-/maxCount, M4 [24], Bloom filters [11], mergeable summaries [4], algebraic classifiers [22], Apache Flink's interface for `AggregateFunction` [14]) and strike the best balance between expressiveness and efficiency. For strictly in-order streams, performance differences between these two classes are minor. The fastest algorithms for associative operators take only $O(1)$ time per window change, without requiring invertibility, commutativity, or other properties [29, 32].

Table 1: Aggregation operators.

	Invertible	Associative	Commutative
Sum-like: sum, count, mean, geomean, stddev, ...	✓	✓	✓
Collect-like: collect list, concat strings, ith-youngest, ...	✓	✓	×
Max-like: max, min, argMax, maxCount, M4 [24], ...	×	✓	?
Mergeable sketch [4]: Bloom, CountMin, HyperLogLog, algebraic classifiers [22], ...	×	✓	✓

In reality, however, out-of-order streams are the norm [5], with support provided on many stream processing platforms (e.g., [6, 7, 14, 37]). At the level of aggregation operators, solutions to out-of-order arrivals fall into two broad categories: **(i)** Solutions that buffer data items long enough to reorder them for use with an in-order aggregation operator. This approach is simple but prone to high latency, making it intolerable for applications that require real-time or fine-grained queries, e.g., when seeking millisecond responsiveness to avert risks. **(ii)** Solutions that eagerly incorporate data items into the aggregation data structure as soon as they arrive. As such, the aggregation is always updated and ready for queries, but this requires out-of-order handling in the aggregation framework. Its main advantage is more controlled latency. The best known solution uses an augmented balanced tree (e.g., the red-black tree), costing $O(\log n)$ time per insert or evict, where n is the window size.

Still, out-of-order streams come in many flavors: Clock skew and network delays cause random, but mostly small, delays [31]. Batching causes frequent short spikes of moderate delays [16]. Failure followed by subsequent recovery can cause rare bursts of large delays [25]. In all these scenarios, the $O(\log n)$-time bound stands in stark contrast with $O(1)$ for the in-order setting. In practice, this gap is found to translate to several-fold differences in latency and throughput. This work sets out to bridge this difference.

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339499

In another line of work, because many early streaming systems use sliding-window algorithms that are too slow for large window sizes n, they encourage users to specify coarse-grained windows, thereby reducing the effective n [15, 26, 27, 34]. This puts the burden on the programmers to trade off accuracy against time. While there are scenarios where coarse-grained windows suffice, only fine-grained windows can capture detailed behavior of a stream. Furthermore, fixing a coarse granularity up-front precludes finer granularities in ad-hoc queries. This work aims to reduce the technology limitations that motivate such manual trade-off, so users can pick the window granularity more on the basis of application requirements. That said, when desired, our solution can also use coarse-grained windows.

This paper introduces the finger B-tree aggregator (FiBA), a novel algorithm that *optimally* aggregates sliding windows on out-of-order streams and in-order streams alike. Each insert or evict takes amortized $O(\log d)$ time, where the *out-of-order distance* d is the distance from the inserted or evicted value to the closer end of the window (see Theorem 5 for a more formal statement). This means $O(1)$ for in-order streams, nearly $O(1)$ for slightly out-of-order streams, and never more than $O(\log n)$ for even severely out-of-order streams. FiBA requires $O(n)$ space and takes $O(1)$ time for a whole-window query. It is as general as the prior state-of-the-art, supporting variable-sized windows and querying of any subwindow while only requiring associativity from the operator. To our knowledge, no existing out-of-order algorithms can achieve both this time bound and this level of generality.

In broad strokes, the heart of our algorithm is a specially-designed B-tree with the following technical features:

- ▷ *Lean finger-searching.* By maintaining minimal fingers (pointers) to the start and end of the tree, FiBA can locate any value to insert or evict in $O(\log d)$ worst-case time.
- ▷ *Constant-time rebalancing.* By carefully selecting and adapting a specific variant of B-trees, the tree can store values in both internal and leaf nodes (fully utilizing the space), and can be rebalanced in $O(1)$ amortized time via a somewhat lazy strategy. Note that the more standard B-tree variants with, for example, `MAX_ARITY`$= 2\cdot$`MIN_ARITY`-1 and eager rebalancing, while optimized for top-down operations, do not have the same guarantees.
- ▷ *Position-aware partial aggregates.* By devising position-aware partial aggregates and a corresponding algorithm, FiBA keeps the cost of aggregate repairs at most that of search and rebalance.

Furthermore, FiBA is optimal and supports window sharing. We prove that FiBA has the best asymptotic running time possible, showing that for `insert` and `evict` with out-of-order distance up to d, in the worst case, an operation must cost at least amortized $\Omega(\log d)$. We also show how FiBA can support window sharing with query time logarithmic in the subwindow size and the distance from the largest window's boundaries. Here, the space required is $O(n_{\max})$, where $n_{\max}$ is the size of the largest window.

Our experiments confirm the theoretical findings and show that FiBA performs well in practice. This paper reports results with both synthetic and real data, with implementations in both C++ and Java, and with comparisons against Flink [14], RA [33], TwoStacks and DABA [32], and (new) Scotty [34]. For large windows over slightly out-of-order streams, FiBA yields up to 4.9× higher throughput than existing algorithms. For strictly in-order streams (i.e., FIFO), FiBA demonstrates constant-time performance and, depending on the implementation language/platform, can be slower or faster than specialized solutions for in-order streams.

Overall, FiBA makes out-of-order sliding-window aggregation less resource-hungry and more responsive, enabling it to be used in situations where it was previously deemed too expensive.

2. OOO SWAG AND LOWER BOUND

Consider a data stream where each value carries a logical time in the form of a timestamp. Throughout, denote a timestamped value as $\left[\begin{smallmatrix} t \\ v \end{smallmatrix}\right]$ or $t:v$ in inline rendering. For example, $\left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right]$ or 2.3:5 is the value 5 at logical time 2.3. The examples in this paper use real numbers for timestamps, but our algorithms do not depend on any of their properties besides total ordering, so, e.g., they work just as well with date/time representations.

It may seem intuitive to assume that values in such a stream arrive in nondecreasing order of time (in order). However, due to clock drift and disparate latency in computation and communication, among other factors, values in a stream often arrive in a different order than their timestamps. Such a stream is said to have *out-of-order* (OoO) arrivals—there exists a later-arriving value that has an earlier logical time than a previously-arrived value.

Our goal in this paper is to maintain the aggregate value of a time-ordered sliding window in the face of out-of-order arrivals. To motivate our formulation below, consider the following example, which maintains the max and the maxcount, i.e., the number of times the max occurs in a 5-second sliding window.

$\left[\begin{smallmatrix} 2.0 \\ 4 \end{smallmatrix}\right], \left[\begin{smallmatrix} 3.0 \\ 3 \end{smallmatrix}\right], \left[\begin{smallmatrix} 4.0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.0 \\ 4 \end{smallmatrix}\right]$ max 4, maxcount 2

Initially, the values $4, 3, 0, 4$ arrive in the same order as their associated timestamps $2.0, 3.0, 4.0, 6.0$. The maximum value is 4, and maxcount is 2 because 4 occurs twice. When stream values arrive in order, they are simply appended. For instance, when $\left[\begin{smallmatrix} 6.5 \\ 4 \end{smallmatrix}\right]$ arrives, it is inserted at the end:

$\left[\begin{smallmatrix} 2.0 \\ 4 \end{smallmatrix}\right], \left[\begin{smallmatrix} 3.0 \\ 3 \end{smallmatrix}\right], \left[\begin{smallmatrix} 4.0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.0 \\ 4 \end{smallmatrix}\right], \mathbf{\left[\begin{smallmatrix} 6.5 \\ 4 \end{smallmatrix}\right]}$ max 4, maxcount 3

But when values arrive out-of-order, they must be inserted into the appropriate spots to keep the window time-ordered. For instance, when $\left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right]$ arrives, it is inserted between timestamps 2.0 and 3.0:

$\left[\begin{smallmatrix} 2.0 \\ 4 \end{smallmatrix}\right], \mathbf{\left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right]}, \left[\begin{smallmatrix} 3.0 \\ 3 \end{smallmatrix}\right], \left[\begin{smallmatrix} 4.0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.0 \\ 4 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.5 \\ 4 \end{smallmatrix}\right]$ max 5, maxcount 1

As for eviction, stream values are usually removed from a window in order. For instance, if the watermark progresses to 7.2, that causes $\left[\begin{smallmatrix} 2.0 \\ 4 \end{smallmatrix}\right]$ to be evicted from the front of the 5-second window:

$\cancel{\left[\begin{smallmatrix} 2.0 \\ 4 \end{smallmatrix}\right]}, \left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right], \left[\begin{smallmatrix} 3.0 \\ 3 \end{smallmatrix}\right], \left[\begin{smallmatrix} 4.0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.0 \\ 4 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.5 \\ 4 \end{smallmatrix}\right]$ max 5, maxcount 1

Notice that, in general, eviction cannot always be accomplished by simply inverting the aggregation value. For instance, when the watermark reaches 7.7, evicting $\left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right]$ cannot be done by "subtracting off" the value 5 from the current aggregation value. The algorithm needs to efficiently discover the new max 4 and maxcount 2:

$\cancel{\left[\begin{smallmatrix} 2.3 \\ 5 \end{smallmatrix}\right]}, \left[\begin{smallmatrix} 3.0 \\ 3 \end{smallmatrix}\right], \left[\begin{smallmatrix} 4.0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.0 \\ 4 \end{smallmatrix}\right], \left[\begin{smallmatrix} 6.5 \\ 4 \end{smallmatrix}\right]$ max 4, maxcount 2

Monoids. Monoids capture a large class of commonly used aggregations [12, 33]. A *monoid* is a triple $\mathcal{M} = (S, \otimes, \mathbf{1})$, where $\otimes : S \times S \to S$ is a binary associative operator on S, with $\mathbf{1}$ being its identity element. Notice that $\otimes$ only needs to be associative; while it is sometimes also commutative or invertible, neither of those additional properties is required. For example, to express max and maxcount as a monoid, if m and c are the max and maxcount, then

$$\langle m_1, c_1\rangle \otimes_{\mathsf{max,maxcount}} \langle m_2, c_2\rangle = \begin{cases} \langle m_1, c_1\rangle & \text{if } m_1 > m_2 \\ \langle m_2, c_2\rangle & \text{if } m_1 < m_2 \\ \langle m_1, c_1 + c_2\rangle & \text{if } m_1 = m_2 \end{cases}$$

Since $\otimes$ is associative, no parentheses are needed for repeated application. When the context is clear, we omit $\otimes$, e.g., writing qstu for q ⊗ s ⊗ t ⊗ u.

OoO SWAG. This paper is concerned with maintaining an aggregation on a time-ordered sliding window where the aggregation

operator can be expressed as a monoid. This can be formulated as an abstract data type (ADT) as follows:

Definition 1. Let $(\otimes, \mathbf{1})$ be a binary operator from a monoid and its identity. The *out-of-order sliding-window aggregation* (OoO SWAG) ADT is to maintain a time-ordered sliding window $\left[\begin{smallmatrix} t_1 \\ v_1 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_n \\ v_n \end{smallmatrix}\right]$, $t_i < t_{i+1}$, supporting the following operations:

— `insert(t: Time, v: Agg)` checks whether t is already in the window, i.e., whether there is an i such that $t = t_i$. If so, it replaces $\left[\begin{smallmatrix} t_i \\ v_i \end{smallmatrix}\right]$ by $\left[\begin{smallmatrix} t_i \\ v_i \otimes v \end{smallmatrix}\right]$. Otherwise, it inserts $\left[\begin{smallmatrix} t \\ v \end{smallmatrix}\right]$ into the window at the appropriate location.

— `evict(t: Time)` checks whether t is in the window, i.e., whether there is an i such that $t = t_i$. If so, it removes $\left[\begin{smallmatrix} t_i \\ v_i \end{smallmatrix}\right]$ from the window. Otherwise, it does nothing.

— `query(): Agg` combines the values in time order using the $\otimes$ operator. In other words, it returns $v_1 \otimes \ldots \otimes v_n$ if the window is non-empty, or $\mathbf{1}$ if empty.

Lower Bound. For in-order streams, the SWAG operations take $O(1)$ time per operation [32]. The problem becomes more difficult when the stream has out-of-order arrivals. This paper shows that to handle out-of-order distance up to d, the amortized cost of a OoO SWAG operation in the worst case must be at least $\Omega(\log d)$.

Theorem 1. *Let $m, d \in \mathbb{Z}$ be given such that $m \geq 1$ and $0 \leq d \leq m$. For any OoO SWAG algorithm, there exists a sequence of $3m$ operations, each with out-of-order distance at most d, for which the algorithm requires a total of at least $\Omega(m \log(1 + d))$ time.*

Our proof (in the appendix) shows this in two steps: (i) it establishes a sorting lower bound for permutations on m elements with out-of-order distance at most d; and (ii) it gives a reduction proving that maintaining OoO SWAG is no easier than sorting such permutations.

Orthogonal Techniques. OoO SWAG operations are designed to work well with other stream aggregation techniques. The `insert(t, v)` operation supports the case where t is already in the window, so it works with pre-aggregation schemes such as window panes [27], paired windows [26], Cutty [15], or Scotty [34]. For instance, for a 5-hour sliding window that advances in 1-minute increments, the logical times can be rounded to minutes, leading to more cases where t is already in the window. The `evict(t)` operation accommodates the case where t is not the oldest time in the window, so it works with streaming systems that use retractions [2, 5, 6, 9, 13, 17, 28, 37]. Neither `insert(t, v)` nor `evict(t)` is limited to values of t that are near either end of the window, so they work in the general case, not just in cases where the out-of-order distance is bounded by buffer sizes or low watermarks.

Query Sharing. Definition 1 does not support query sharing. But query sharing can be accommodated by adding a range query:

— `query(t_from: Time, t_to: Time): Agg` aggregates exactly the values in the window whose times fall between $t_{\texttt{from}}$ and $t_{\texttt{to}}$. That is, it returns $v_{i_{\texttt{from}}} \otimes \ldots \otimes v_{i_{\texttt{to}}}$, where $i_{\texttt{from}}$ is the largest i such that $t_{\texttt{from}} \leq t_{i_{\texttt{from}}}$ and $i_{\texttt{to}}$ is the smallest i such that $t_{i_{\texttt{to}}} \leq t_{\texttt{to}}$. If the subrange contains no values, it returns $\mathbf{1}$.

In these terms, this paper aims to construct efficient OoO SWAG for arbitrary monoids.

3. FINGER B-TREE AGGREGATOR (FIBA)

This section introduces our algorithm gradually, giving intuition along the way. It begins by describing a basic algorithm (Section 3.1) that utilizes a B-tree augmented with aggregates. This takes $O(\log n)$ time for each `insert` or `evict`. Reducing the time complexity below $\log n$ requires further observations. This is explored intuitively in Section 3.2 with details fleshed out in Section 3.3.

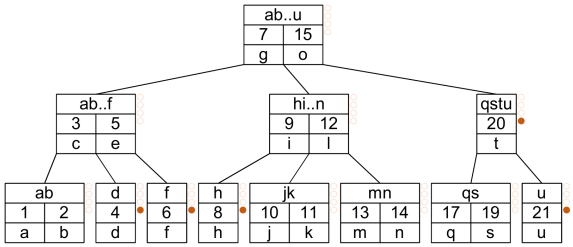

Figure 1: Classic B-tree augmented with aggregates (top cell of each node). The notation for stored aggregates omits $\otimes$, e.g., qstu is shorthand for the value of q $\otimes$ s $\otimes$ t $\otimes$ u.

3.1 Basic Algorithm: Augmented B-Tree

One way to implement OoO SWAG is to start with a classic B-tree with timestamps as keys and augment that tree with aggregates. This is a baseline implementation, which will be built upon. Even though at this stage, any balanced trees can be used, we chose the B-tree because it is well-studied and its customizable fan-out degree enables trading intra-node costs against inter-node costs.

There are many B-tree variations. The range of permissible *arity*, or fan-out degree of a node, is controlled by two parameters `MIN_ARITY` and `MAX_ARITY`. While `MIN_ARITY` can be any integer greater than 1, most B-tree variations require that `MAX_ARITY` be at least $2 \cdot$ `MIN_ARITY` $- 1$. Let $a(y)$ denote the arity of node y, and just a in clear context. Then a B-tree obeys the following *size invariants*:

- For a non-root node y, `MIN_ARITY` $\leq a(y)$; for the root, $2 \leq a$.
- For all nodes, $a \leq$ `MAX_ARITY`.
- All nodes have $a - 1$ timestamps and values $\left[\begin{smallmatrix} t_0 \\ v_0 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_{a-2} \\ v_{a-2} \end{smallmatrix}\right]$.
- All non-leaf nodes have a child pointers $z_0, \ldots, z_{a-1}$.

Figure 1 illustrates a B-tree augmented with aggregates. In this example, `MIN_ARITY` is 2 and `MAX_ARITY` is $2 \cdot$ `MIN_ARITY` $= 4$. Consequently, all nodes have 1–3 timestamps and values, and non-leaf nodes have 2–4 children. Each node in the tree contains an aggregate, an array of timestamps and values, and optional pointers to the children. For instance, the root node contains the aggregate ab..u which is shorthand for a $\otimes$ b $\otimes$ .. $\otimes$ u, two timestamps and values $\left[\begin{smallmatrix} 7 \\ g \end{smallmatrix}\right], \left[\begin{smallmatrix} 15 \\ o \end{smallmatrix}\right]$, and pointers to three children. Because we use timestamps as keys, the entries are time-ordered, both within a node and across nodes. We store timestamps in a parent node separating and limiting the time in the subtrees it points to. The tree is always height-balanced and all leaves are at the same depth.

What aggregate to keep in a node? For each node y, the aggregate $\Pi_\uparrow(y)$ stored at that node obeys the *up-aggregation* invariant:

$$\Pi_\uparrow(y) = \Pi_\uparrow(z_0) \otimes v_0 \otimes \Pi_\uparrow(z_1) \otimes \ldots \otimes v_{a-2} \otimes \Pi_\uparrow(z_{a-1})$$

By a standard inductive argument, $\Pi_\uparrow(y)$ is the aggregation of the values inside the subtree rooted at y. This means the `query()` operation can simply return the aggregation value at the root (`root.agg`).

The operations `insert(t, v)` or `evict(t)` first search for the node where t belongs. Second, they locally insert or evict at that node, updating the aggregate stored at that node. Then, they rebalance the tree, walking from that node towards the root as necessary to fix any size invariant violations, while also repairing aggregate values along the way. Finally, they repair any remaining aggregate values not repaired during rebalancing, starting above the node where rebalancing topped out and visiting all ancestors up to the root.

Theorem 2. *In a classic B-tree augmented with aggregates, if it stores $\left[\begin{smallmatrix} t_1 \\ v_1 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_n \\ v_n \end{smallmatrix}\right]$, the operation `query()` returns $v_1 \otimes \ldots \otimes v_n$.*

Theorem 3. *In a classic B-tree augmented with aggregates, the operation `query()` costs at most $O(1)$ time and operations `insert(t, v)` or `evict(t)` take at most $O(\log n)$ time.*

The theorems follow directly from the up-aggregation invariant and the fact that the tree height is $O(\log n)$ [10, 18, 21].

3.2 Breaking the $O(\log n)$ Barrier

The basic algorithm just described requires $O(\log n)$ time per update. To improve upon the time complexity, we now discuss the bottlenecks in the basic algorithm and outline a plan to resolve them.

In the basic algorithm, the `insert(t, v)` and `evict(t)` operations involve four steps: (1) search for the node where t belongs; (2) locally insert or evict; (3) rebalance to repair size invariants; and (4) repair remaining aggregation invariants. The local insertion or eviction takes constant time, as does the `query()`. But each of the steps for search, rebalance, and repair takes up to $O(\log n)$ time. Hence, these are the bottleneck steps and will be improved upon as follows:

(i) Maintaining "fingers" to the leftmost and rightmost leaves reduces the search complexity to $O(\log d)$, where d is the distance to the closer end of the sliding-window boundary. For the in-order or nearly in-order case, this means constant-time search.

(ii) By choosing an appropriate `MAX_ARITY` and a somewhat lazy strategy for rebalancing, we can prove that rebalance takes no more than constant time in the amortized sense. This means that for any operation, the cost to restore the proper tree structure amounts to constant, regardless of out-of-order distance.

(iii) By introducing position-dependent aggregates, we will ensure that repairs to the aggregate values are made only to nodes along the search path or involved in restructuring. This means that the repairs cost no more than the cost of search and rebalance.

Our novel FiBA algorithm combines these ideas to implement OoO SWAG in $O(\log d)$ time, $d \le n$. Below, we describe how they will be implemented intuitively, leaving details to Section 3.3.

Search. In classic B-trees, a search starts at the root and ends at the node being searched for, henceforth called y. Often, y is a leaf, so the search visits $O(\log n)$ nodes. However, instead of starting at the root, one can start at the left-most or right-most leaf in the tree. This requires pointers to the corresponding leaf, henceforth called the left and right fingers [19]. In addition, we keep a parent pointer at each node. Hence, the search can start at the nearest finger, walk up to the nearest common ancestor of the finger and y, and walk down from there to y. The resulting algorithm runs in $O(\log d)$, where d is the distance from the nearest end of the window—or more precisely, d is the number of timed values from y to the nearest end of the window.

Rebalance. Insertions and evictions can cause nodes to overflow or underflow, violating the size invariants. There are two popular correction strategies: either before or after the fact. The before-the-fact strategy ensures that ancestors of the affected node are not at risk of overflowing or underflowing by preventive rebalancing (e.g., [18]). The after-the-fact strategy first performs the local insert or evict, then repairs any resulting overflow or underflow to ensure the size invariants hold again by the end of the entire insert or evict operation. We adopt the after-the-fact strategy, as it is amortized constant as long as $\texttt{MAX\_ARITY} \ge 2 \cdot \texttt{MIN\_ARITY}$ (see Lemma 9, adapted from [21]). For simplicity, we use $\texttt{MAX\_ARITY} = 2 \cdot \texttt{MIN\_ARITY}$. The amortized cost is $O(1)$ as rebalancing rarely goes all the way up the tree. The worst-case cost is $O(\log n)$, bounded by the tree height.

Repair. The basic algorithm stores at each node y the up-aggregate $\Pi_{\uparrow}(y)$, i.e., the partial aggregate of the subtree under y. This is problematic, because it means that an insertion or eviction at a node z, usually a leaf, affects the partial aggregates stored in all ancestors of z—which is the entire path up to the root. To circumvent this issue, we need an arrangement of aggregates that can be repaired by traversing to a finger, *without* always traversing to the root. For this, each node stores a kind of partial aggregate suitable for its position

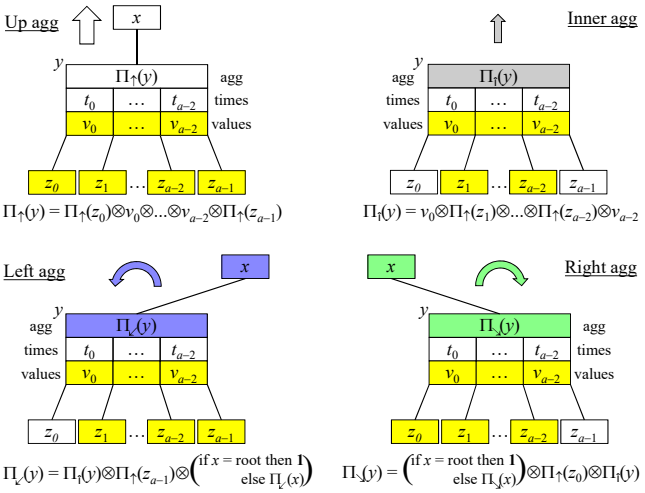

Figure 2: Partial aggregates definitions.

in the tree. A further consequence is the root no longer contains the aggregate of the whole tree, and we need to ensure that `query()` can be answered by combining partial aggregates at the left finger, the root, and the right finger. To meet these requirements, we define four kinds of partial aggregates in Figure 2. They are used in a B-tree according to the following *aggregation invariants*:

▷**Non-spine nodes store the up-aggregate $\Pi_{\uparrow}$.** Such a node is neither a finger nor an ancestor of a finger. This aggregate must be repaired whenever the subtree below it changes. Figure 3(A) shows nodes with up-aggregates in white, light blue, or light green. For example, the center child of the root contains the aggregate hijklmn, comprising its entire subtree.

▷**The root stores the inner aggregate Π_{I}.** This aggregate is only affected by changes to the inner part of the tree, and not by changes below the left-most or right-most child of the root. Figure 3(A) shows the inner parts of the tree in white and the root in gray.

▷**Non-root nodes on the left spine store the left aggregate $\Pi_{\swarrow}$.** For a given node y, the left aggregate encompasses all nodes under the left-most child of the root except for y's left-most child z_0. When a change occurs below the left-most child of the root, the only aggregates that need to be repaired are those on a traversal up to the left spine and then down to the left finger. Figure 3(A) shows the left spine in dark blue and nodes affecting it in light blue. For example, the node in the middle of the left spine contains the aggregate cdef, comprising the left subtree of the root except for the left finger.

▷**Non-root nodes on the right spine store the right aggregate $\Pi_{\searrow}$.** This is symmetric to the left aggregate $\Pi_{\swarrow}$. When a change occurs below the right-most child of the root, only aggregates on a traversal to the right finger are repaired. Figure 3(A) shows the right spine in dark green and nodes affecting it in light green.

3.3 Maintaining Finger B-Tree OoO SWAG

We will now flesh out FiBA with a focus on how `insert` and `evict` update the tree to maintain the size invariants from Section 3.1 and the aggregation invariants from Section 3.2.

Throughout, our running time discussion will be based on coin counting. This aligns with our formal amortized analysis (Lemma 9) using the accounting method [18], which can be thought of as keeping coins at tree nodes. A coin is *spent* to cover the cost of one internal $O(1)$-time local-modification operation (split, merge, or move). The algorithm (conceptually) *bills* each `insert(t, v)` and `evict(t)` to make up the difference. Excess coins are *refunded* to maintain the appropriate balance. As the analysis will show, each

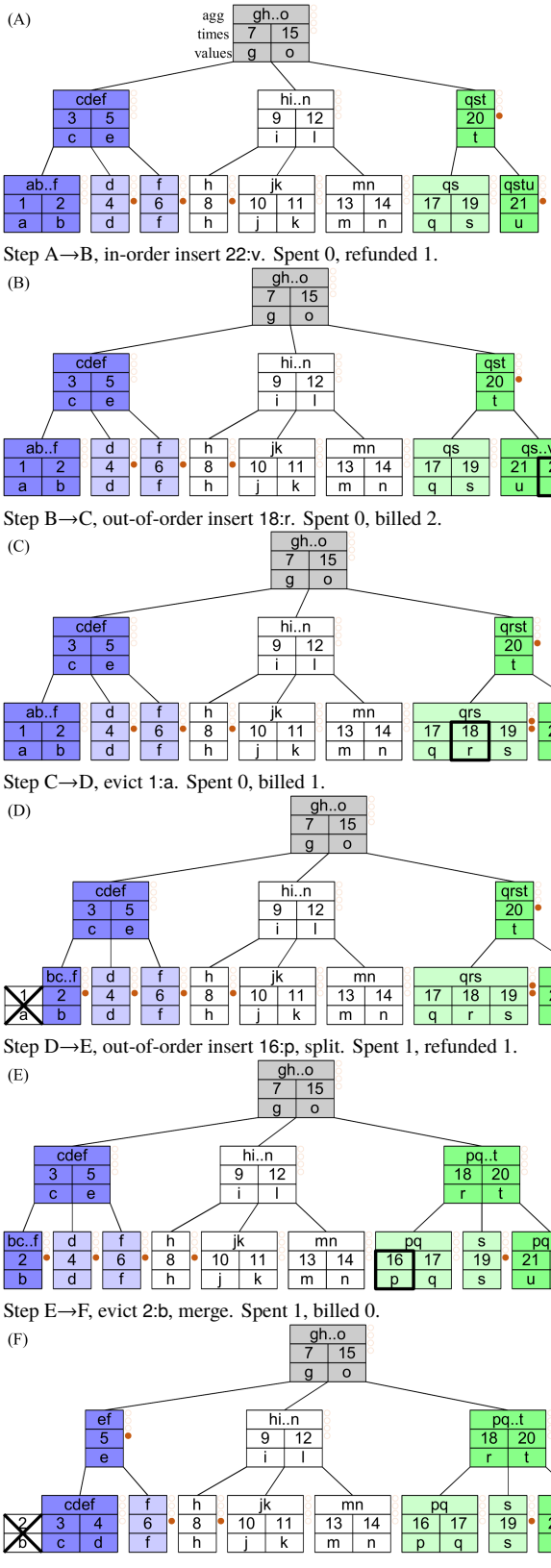

Figure 3: Finger B-tree with aggregates: example.

Figure 4: Finger B-tree height increase and split.

Figure 5: Finger B-tree move.

Figure 6: Finger B-tree merge and height decrease.

`insert`(t, v) or `evict`(t) is never billed more than 2 coins, hence rebalancing is amortized $O(1)$ time. To visualize this accounting, coins are rendered as little golden circles next to tree nodes.

All interesting cases of tree maintenance are illustrated in Figures 3–6. Each state, for instance (A), shows a tree with aggregates and coins. Each step, for instance A→B, shows an insert or evict, illustrating how it affects the tree, partial aggregates, and coins.

- In Figure 3, Step A→B is an in-order insert without rebalance, which only affects the aggregate at a single node, the right finger.
- Step B→C is an out-of-order insert without rebalance, affecting aggregates on a walk to the right finger.
- Step C→D is an in-order evict without rebalance, affecting the aggregate at a single node, the left finger.
- Step D→E is an out-of-order insert to a node with arity $a = 2 \cdot$ `MIN_ARITY`, causing an overflow; rebalancing splits it.
- Step E→F is an evict from a node with $a =$ `MIN_ARITY`, causing the node to underflow; rebalancing merges it with its neighbor.
- In Figure 4, Step G→H is an insert that causes nodes to overflow

```
1  fun query(): Agg
2    if root.isLeaf()
3      return root.agg
4    return leftFinger.agg ⊗ root.agg ⊗ rightFinger.agg
5
6  fun insert(t: Time, v: Agg)
7    node ← searchNode(t)
8    node.localInsertTimeAndValue(t, v)
9    top, hit_left, hit_right ← rebalanceForInsert(node)
10   repairAggs(top, hit_left, hit_right)
11
12 fun evict(t: Time)
13   node ← searchNode(t)
14   found, idx ← node.localSearch(t)
15   if found
16     if node.isLeaf()
17       node.localEvictTimeAndValue(t)
18       top, hit_left, hit_right ← rebalanceForEvict(node, null)
19     else
20       top, hit_left, hit_right ← evictInner(node, idx)
21     repairAggs(top, hit_left, hit_right)
22
23 fun repairAggs(top: Node, hit_left: Bool, hit_right: Bool)
24   if top.hasAggUp()
25     while top.hasAggUp()
26       top ← top.parent
27       top.localRepairAgg()
28   else
29     top.localRepairAgg()
30   if top.leftSpine or top.isRoot() and hit_left
31     left ← top
32     while not left.isLeaf()
33       left ← left.getChild(0)
34       left.localRepairAgg()
35   if top.rightSpine or top.isRoot() and hit_right
36     right ← top
37     while not right.isLeaf()
38       right ← right.getChild(right.arity - 1)
39       right.localRepairAgg()
40 fun rebalanceForInsert(node: Node): Node×Bool×Bool
41   hit_left, hit_right ← node.leftSpine, node.rightSpine
42   while node.arity > MAX_ARITY
43     if node.isRoot()
44       heightIncrease()
45       hit_left, hit_right ← true, true
46     split(node)
47     node ← node.parent
48     hit_left ← hit_left or node.leftSpine
49     hit_right ← hit_right or node.rightSpine
50   return node, hit_left, hit_right
51
52 fun rebalanceForEvict(node: Node, toRepair: Node)
53                      : Node×Bool×Bool
54   hit_left, hit_right ← node.leftSpine, node.rightSpine
55   if node = toRepair
56     node.localRepairAggIfUp()
57   while not node.isRoot() and node.arity < MIN_ARITY
58     parent ← node.parent
59     nodeIdx, siblingIdx ← pickEvictionSibling(node)
60     sibling ← parent.getChild(siblingIdx)
61     hit_right ← hit_right or sibling.rightSpine
62     if sibling.arity ≤ MIN_ARITY
63       node ← merge(parent, nodeIdx, siblingIdx)
64       if parent.isRoot() and parent.arity = 1
65         heightDecrease()
66       else
67         node ← parent
68     else
69       move(parent, nodeIdx, siblingIdx)
70       node ← parent
71     if node = toRepair
72       node.localRepairAggIfUp()
73     hit_left ← hit_left or node.leftSpine
74     hit_right ← hit_right or node.rightSpine
75   return node, hit_left, hit_right
```

Figure 7: Finger B-Tree with aggregates: algorithm.

```
76 fun evictInner(node: Node, idx: Int): Node×Bool×Bool
77   left, right ← node.getChild(idx), node.getChild(idx+1)
78   if right.arity > MIN_ARITY
79     leaf, t_leaf, v_leaf ← oldest(right)
80   else
81     leaf, t_leaf, v_leaf ← youngest(left)
82   leaf.localEvictTimeAndValue(t_leaf)
83   node.setTimeAndValue(idx, t_leaf, v_leaf)
84   top, hit_left, hit_right ← rebalanceForEvict(leaf, node)
85   if top.isDescendent(node)
86     while top ≠ node
87       top ← top.parent
88       hit_left ← hit_left or top.leftSpine
89       hit_right ← hit_right or top.rightSpine
90       top.localRepairAggIfUp()
91   return top, hit_left, hit_right
```

Figure 8: Finger B-Tree evict inner: algorithm.

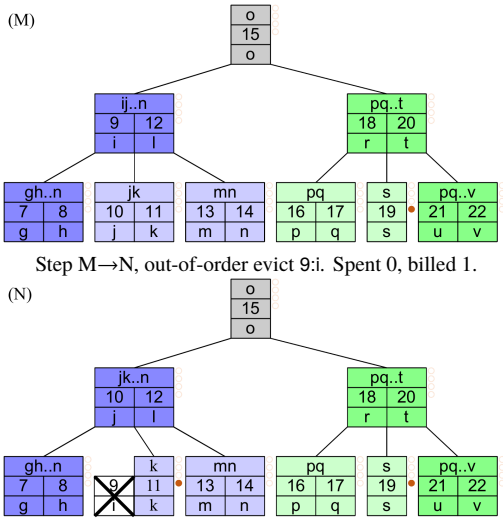

Figure 9: Finger B-tree evict inner: example.

all the way up to the root, causing a height increase followed by splitting the old root. This affects aggregates on all split nodes and on both spines.

- In Figure 5, Step I→J is an evict that causes first an underflow that is fixed by a merge, and then an underflow at the next level where the neighbor node is too big to merge. The algorithm repairs the size invariant with a move of a child and a timed value from the neighbor, touching aggregates on all nodes affected by rebalancing plus a walk to the left finger.
- In Figure 6, Step K→L is an evict that causes nodes to underflow all the way up to the root, causing a height decrease. This affects aggregates on all merged nodes and on both spines.

Figure 7 shows most of the algorithm, excluding only `evictInner`, which will be presented later. While rebalancing always works bottom-up, aggregate repair works in the direction of the partial aggregates: either up for up-agg or inner-agg, or down for left-agg or right-agg. Our algorithm piggybacks the repair of up-aggs onto the local insert or evict and onto rebalancing, and then repairs the remaining aggregates separately. To facilitate the handover from the piggybacked phase to the dedicated phase of aggregate repair, the rebalancing routines return a triple $\langle top, hit_{left}, hit_{right} \rangle$, for instance, in Line 9. Node *top* is where rebalancing topped out, and if it has an up-agg, it is the last node whose aggregate has already

been repaired. Booleans $hit_{\texttt{left}}$ and $hit_{\texttt{right}}$ indicate whether rebalancing affected the left or right spine, determining whether aggregates on the respective spine have to be repaired.

Figure 8 shows function *evictInner*. To evict something from an inner node, Line 82 evicts a substitute from a leaf instead, and Line 83 writes that substitute over the evicted slot. Function *evictInner* creates an obligation to repair an extra node during rebalancing, handled by parameter *toRepair* on Line 52 in the same figure.

THEOREM 4. *In a finger B-tree with aggregates that contains* $\left[\begin{smallmatrix} t_1 \\ v_1 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_n \\ v_n \end{smallmatrix}\right]$, *operation* query() *returns* $v_1 \otimes \ldots \otimes v_n$.

PROOF. If the root is a leaf, the root stores an inner aggregate, representing all the values. Otherwise, correctness follows directly from the aggregation invariants. □

THEOREM 5. *In a finger B-tree with aggregates,* query() *costs* $O(1)$ *time, and* insert(t, v) *and* evict(t) *take time* $T_{search} + T_{rebalance} + T_{repair}$, *where*

- T_{search} *is* $O(\log d)$, *with* d *being the distance to the start or end of the window, whichever is closer;*
- $T_{rebalance}$ *is amortized* $O(1)$ *and worst-case* $O(\log n)$*; and*
- T_{repair} *is* $O(T_{search} + T_{rebalance})$.

PROOF. The *query*() operation performs at most *two* $\otimes$ operations, hence running in $O(1)$ time. The search cost T_{search} is bounded as follows: Let y_0 be the node at the finger where search begins and define y_{i+1} as the parent of y_i. This forms a sequence of spinal nodes on which searching takes place. Remember $\mu = \textit{MIN\_ARITY}$ is constant. Because the subtree at y_i has $\Omega(\mu^i)$ keys and the search key is at distance d, the key must belong in the subtree rooted at some y_{i^*}, where $i^* = O(\log d)$. Thus, it takes i^* steps to walk up the spine and at most another i^* to locate the spot in the subtree as all leaves are at the same depth, bounding T_{search} by $O(\log_\mu d)$. Lemma 9 in the appendix gives the rebalance cost $T_{\text{rebalance}}$. Finally, by the aggregation invariants, a partial aggregation is affected only if it is along the search path or involved in rebalancing. Therefore, the number of affected nodes that requires repairs is bounded by $\Theta(T_{\text{search}} + T_{\text{rebalance}})$. Treating μ as bounded by a constant, T_{repair} is $O(T_{\text{search}} + T_{\text{rebalance}})$, concluding the proof. □

4. WINDOW SHARING

This section explains how to use a single finger B-tree to efficiently answer queries on subwindows of different sizes on the fly. Applications are numerous. One common example is simple anomaly detection that compares two related aggregates: one on a large window representing normal stable behavior and the other on a small window representing recent behavior. An alert is triggered when the aggregates differ substantially. Whereas in this example, the sizes of the windows are known ahead of time, in other applications—e.g., interactive data exploration—queries can be ad hoc.

We implement window sharing via range queries, as defined at the end of Section 2. This has many benefits. The window contents need to be saved only once regardless of how many subwindows are involved. Thus, each insert or evict needs to be performed only once on the largest window. This approach can accommodate an arbitrary number of shared window sizes. For instance, many users can register queries over different window sizes. Importantly, queries can be ad hoc and interactive, which would otherwise not be possible to support using multiple fixed instances. Furthermore, the range-query formulation also accommodates the case where the window boundary is not the current time ($t_{\texttt{to}} \neq t_{\texttt{now}}$). For instance, it can report results with some time lag dictated by punctuation or low watermarks.

```
1  fun query(t_from: Time, t_to: Time): Agg
2    node_from, node_to ← searchNode(t_from), searchNode(t_to)
3    node_top ← leastCommonAncestor(node_from, node_to)
4    return queryRec(node_top, t_from, t_to)
5
6  fun queryRec(node: Node, t_from: Time, t_to: Time): Agg
7    if t_from = −∞ and t_to = +∞ and node.hasAggUp()
8      return node.agg
9    res ← 1
10   if not node.isLeaf()
11     t_next ← node.getTime(0)
12     if t_from < t_next
13       res = res ⊗ queryRec(node.getChild(0),
14                            t_from,
15                            t_next ≤ t_to ? +∞ : t_to)
16   for i ∈ [0, ..., node.arity - 2]
17     t_i ← node.getTime(i)
18     if t_from ≤ t_i and t_i ≤ t_to
19       res ← res ⊗ node.getValue(i)
20     if not node.isLeaf() and i + 1 < node.arity − 2
21       t_{i+1} ← node.getTime(i + 1)
22       if t_i < t_to and t_from < t_{i+1}
23         res ← res ⊗ queryRec(node.getChild(i + 1),
24                              t_from ≤ t_i ? −∞ : t_from,
25                              t_{i+1} ≤ t_to ? +∞ : t_to)
26   if not node.isLeaf()
27     t_curr ← node.getTime(node.arity - 2)
28     if t_curr < t_to
29       res = res ⊗ queryRec(node.getChild(node.arity - 1),
30                            t_from ≤ t_curr ? −∞ : t_from,
31                            t_to)
32   return res
```

Figure 10: Range query algorithm.

To answer the range query *query*$(t_{\texttt{from}}, t_{\texttt{to}})$, the algorithm, shown in Figure 10, uses recursion starting from the least-common ancestor node whose subtree encompasses the queried range. The main technical challenge is to avoid making spurious recursive calls. Because the nodes already store partial aggregates, the algorithm should only recurse into a node's children if the partial aggregates cannot be used directly. Specifically, we aim for the algorithm to invoke at most two chains of recursive calls, one visiting ancestors of $node_{\texttt{from}}$ and the other visiting ancestors of $node_{\texttt{to}}$. The insight for preventing spurious recursive calls is that one needs information about neighboring timestamps in a node's parent to determine whether the node itself is subsumed by the range. This is passed down the recursive call: whether the neighboring timestamp in the parent is included in the range on the left or right is indicated by $t_{\texttt{from}} = -\infty$ or $t_{\texttt{to}} = +\infty$, respectively.

This strategy alone would have been similar to range query in an interval tree [18], albeit without explicitly storing the ranges. However, our partial aggregate scheme adds another difficulty: not all nodes store agg-up values $\Pi_\uparrow(y)$. Fortunately, any nodes that lack $\Pi_\uparrow(y)$ must be on one of the two recursion chains, because if a query involves spines of the entire window, then those spines coincide with edges of the intersection between the window and the range.

THEOREM 6. *In a finger B-tree with aggregates that contains* $\left[\begin{smallmatrix} t_1 \\ v_1 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_n \\ v_n \end{smallmatrix}\right]$, *the operation* query$(t_{\texttt{from}}, t_{\texttt{to}})$ *returns the aggregate* $v_{i_{\texttt{from}}} \otimes \ldots \otimes v_{i_{\texttt{to}}}$, *where* $i_{\texttt{from}}$ *is the largest* i *such that* $t_{\texttt{from}} \leq t_{i_{\texttt{from}}}$ *and* $i_{\texttt{to}}$ *is the smallest* i *such that* $t_{i_{\texttt{to}}} \leq t_{\texttt{to}}$.

PROOF. By induction, each recursive call returns the aggregate of the intersection between its subtree and the queried range. □

THEOREM 7. *In a finger B-tree with aggregates that contains* $\left[\begin{smallmatrix} t_1 \\ v_1 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_n \\ v_n \end{smallmatrix}\right]$, *the operation* query$(t_{\texttt{from}}, t_{\texttt{to}})$ *takes time* $O(\log d_{\texttt{from}} + \log d_{\texttt{to}} + \log n_{sub})$, *where*

- $i_{\texttt{from}}$ *is the largest index* i *such that* $t_{\texttt{from}} \leq t_{i_{\texttt{from}}}$
- $i_{\texttt{to}}$ *is the smallest index* i *such that* $t_{i_{\texttt{to}}} \leq t_{\texttt{to}}$

- $d_{from} = \min(i_{from}, n - i_{from})$ *and* $d_{to} = \min(i_{to}, n - i_{to})$ *are the distances to the window boundary*
- $n_{sub} = i_{to} - i_{from}$ *is the size of subwindow being queried.*

Proof. Using finger searches, Line 2 takes $O(\log d_{\texttt{from}} + \log d_{\texttt{to}})$. Now the distance from either $node_{\texttt{from}}$ or $node_{\texttt{to}}$ to the least-common ancestor (LCA) is at most $O(\log n_{\texttt{sub}})$. Therefore, locating the LCA takes at most $O(\log n_{\texttt{sub}})$, and so do subsequent recursive calls in `queryRec` that traverse the same paths. □

In particular, when a query ends at the current time (i.e., when $t_{to} = t_{now}$), the theorem says that the query takes $O(\log n_{sub})$ time, where n_{sub} is the size of the subwindow being queried.

5. RESULTS

This section describes experimental analysis with both synthetic and real data, with implementations in both C++ and Java, and with comparisons against current state-of-the-art techniques.

5.1 Synthetic data in C++

We implemented both OoO SWAG variants in C++: the baseline classic B-tree augmented with aggregates and the finger B-tree aggregator (FiBA). We present experiments with competitive min-arity values: 2, 4 and 8. Higher values for min-arity were never competitive in our experiments. Our experiments run outside of any particular streaming framework so we can focus on the aggregation algorithms themselves. Our load generator produces synthetic data items with random integers. The experiments perform rounds of `evict`, `insert`, and `query` to maintain a sliding window that accepts a new data item, evicts an old one, and produces a result each round.

We present results with three aggregation operators, each representing a category of computational cost. The operator sum performs an integer sum over the window, and its computational cost is less than that of tree traversals and manipulations. The operator geomean performs a geometric mean over the window. For numerical stability, this requires a floating-point log on insertion and floating-point additions during data structure operations. It represents a middle ground in computational cost. The most expensive operator, bloom, is a Bloom filter [11] where the partial aggregations maintain a bitset of size 2^{14}. It represents aggregation operators whose computational cost dominates the cost of maintaining the SWAG data structure.

We ran all experiments on a machine with an Intel Xeon E5-2697 at 2.7 GHz running Red Hat Enterprise Linux Server 7.5 with a 3.10.0 kernel. We compiled all experiments with `g++` 4.8.5 with optimization level `-O3`.

5.1.1 Varying Distance. We begin by investigating how `insert`'s out-of-order distance affects throughput. The distance varying experiments, Figure 11, maintain a constant-sized window of $n = 2^{22} = 4,194,304$ data items. The x-axis is the out-of-order distance d between the newest timestamp already in the window and the timestamp created by our load generator. Our adversarial load generator pre-populates the window with high timestamps and then spends the measured portion of the experiment producing low timestamps. This regime ensures that after the pre-population with high timestamps, the out-of-order distance of each subsequent insertion is precisely d.

This experiment confirms the prediction of the theory. The classic B-tree's throughput is mostly unaffected by the change in distance, but the finger B-tree's throughput starts out significantly higher. At the smallest values of d, the best finger B-tree outperforms the corresponding classic B-tree by a factor of 3.4× for sum, 2.5× for geomean, and 4.9× for bloom. For larger values of d, the finger B-tree throughput follows a $\log d$ trend. All variants enjoy an uptick in performance when $d = n$, that is, when the distance is the size n of the window. This is a degenerate special case. When $n = d$, the lowest timestamp to evict is always in the left-most node in the tree, so the tree behaves like a last-in first-out (LIFO) stack, and inserting and evicting requires no tree restructuring—$O(1)$ time overall.

The min-arity that yields the best-performing B-tree varies with the aggregation operator. For expensive operators, such as bloom, smaller min-arity trees perform better because they perform fewer partial aggregations inside of a node. Conversely, for cheap operators, such as sum, higher min-arity trees that require fewer rebalance and repair operations perform better. The step-like throughput curves for the finger B-trees is a function of their min-arity: larger min-arity means longer sections where the increased out-of-order distance still affects only a subtree with the same height. When the throughput suddenly drops, the increase in d meant an increase in the height of the affected subtree, causing more rebalances and updates.

5.1.2 Latency. The worst-case latency for both classic and finger B-trees is $O(\log n)$, but we expect finger variants to reduce average latency. The experiments in Figure 12 confirm this expectation. All latency experiments use a window size of $n = 2^{22}$. The top set of experiments uses an out-of-order distance of $d = 0$ and the bottom set uses an out-of-order distance of $d = 2^{20} = 1,048,576$. (We chose the latter distance because it is among the worst-performing in the throughput experiments.) The experimental setup is the same as for the throughput experiments, and the latency is for an entire round of `evict`, `insert`, and `query`. The y-axis is the number of processor cycles for a round, in log scale. Since we used a 2.7 GHz machine, 10^3 cycles take 370 nanoseconds and 10^6 cycles take 370 microseconds. The brown bars show the median latency, the shaded regions show the distribution of latencies, and the black bars are the 99.9th percentile. The range is the minimum and maximum latency.

When the out-of-order distance is 0 and the aggregation operator is cheap or only moderately expensive, the worst-case latency in practice for the classic and finger B-trees is similar. This is expected, as the time is dominated by tree operations, and they are worst-case $O(\log n)$. However, the minimum and median latencies are orders of magnitude better for the finger B-trees. This is also expected, since for $d = 0$, the fingers enable amortized $O(1)$ updates. When the aggregation operator is expensive, the finger B-trees have significantly lower latency as they repair fewer partial aggregates.

With an out-of-order distance of $d = 2^{20}$ and cheap or moderately expensive operators, the classic and finger B-trees have similar latency. This is expected: as d approaches n, the worst-case latency for finger B-trees approaches $O(\log n)$. Again, with expensive operators, the minimum, median, and 99.9th percentile of the finger B-tree with min-arity 2 is orders of magnitude lower than that of classic B-trees. There is, however, a curious effect clearly present in the bloom experiments with finger B-trees, but still observable in the others: min-arity 2 has the lowest latency; it gets worse with min-arity 4, then improves with min-arity 8. Recall that the root may be slimmer than the min-arity. With $d = 2^{20}$, depending on the arity of the root, some aggregation repairs walk almost to the root and then back down a spine while others walk to the root and no further. The former case, which walks twice the height, is more expensive than the latter, which walks just the whole height. The frequency of the expensive case is a function of the window size, tree arity, and out-of-order distance, and these factors do not interact linearly.

5.1.3 FIFO: In-order Data. A special case for FiBA is when $d = 0$; with in-order data, the theoretical results show that FiBA enjoys amortized constant time performance. Figure 13 compares the B-tree-based SWAGs against the state-of-the-art SWAGs optimized for first-in, first-out, completely in-order data. TwoStacks only works on in-order data and is amortized $O(1)$ with worst-case $O(n)$ [3]. DABA also only works on in-order data and is worst-case $O(1)$ [32].

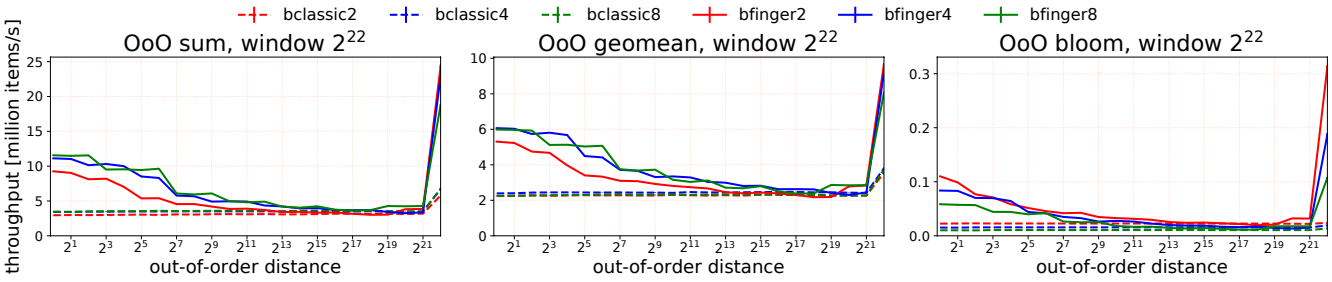

Figure 11: Out-of-order distance experiments.

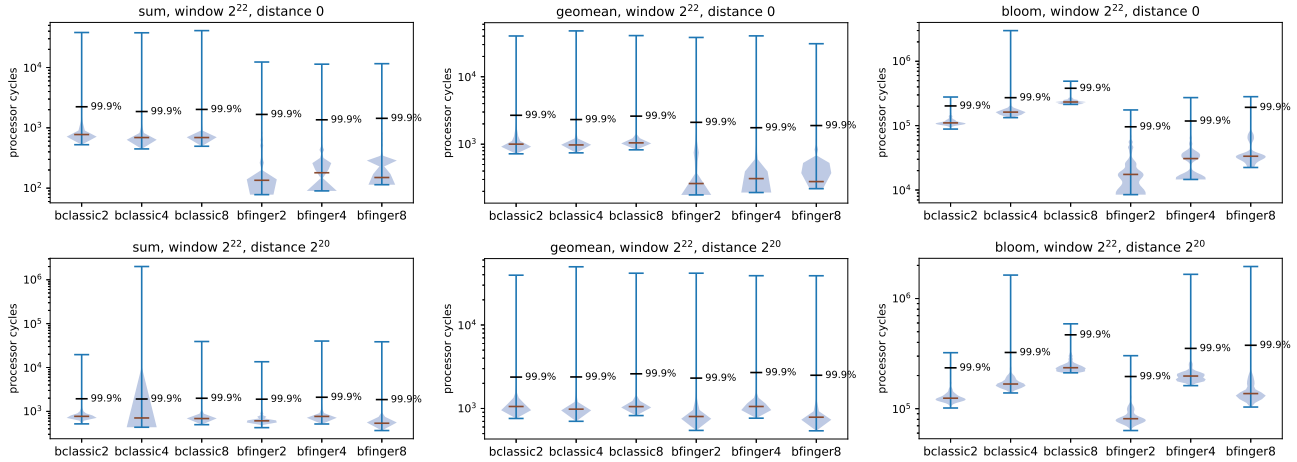

Figure 12: Latency experiments.

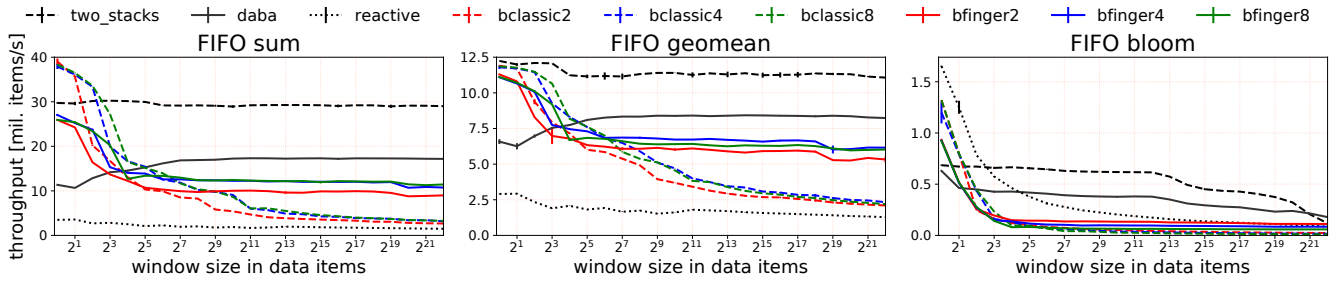

Figure 13: FIFO experiments.

The Reactive Aggregator supports out-of-order evict but requires in-order insert, and is amortized $O(\log n)$ with worst-case $O(n)$ [33]. The x-axis represents the window size n.

TwoStacks and DABA perform as seen in prior work: for most window sizes, TwoStacks with amortized $O(1)$ time has the best throughput. DABA is generally the second best, as it does a little more work on each operation to maintain worst-case constant performance.

The finger B-tree variants demonstrate constant performance as the window size increases. The best finger B-tree variants stay within 30% of DABA for sum and geomean, but are about 60% off of DABA with a more expensive operator like bloom. In general, finger B-trees are able to maintain constant performance with completely in-order data, but the extra work of maintaining a tree means that SWAGs specialized for in-order data consistently outperform them.

The classic B-trees clearly demonstrate $O(\log n)$ behavior as the window size increases. Reactive does demonstrate $O(\log n)$ behavior, but it is only obvious with bloom. For sum and geomean, the fixed costs dominate. Reactive was designed to avoid using pointer-based data structures under the premise that the extra memory accesses would harm performance. To our surprise, this is not true: on our hardware, the extra computation required to avoid pointers ends up costing more. For bloom, Reactive outperforms B-tree based SWAGs because it is essentially a min-arity 1, max-arity 2 tree, thus requiring fewer aggregation operations per node.

5.1.4 Window Sharing. One of the benefits of FiBA is that it supports range queries while maintaining logarithmic performance for queries over that range. Range queries enable window sharing: a single window can support multiple queries over different ranges. An obvious benefit from window sharing is reduced space usage, but we also wanted to investigate its time usage. Figure 14 shows that window sharing did not consistently improve runtime performance.

The experiments maintain two queries: a big window fixed to size 2^{22}, and a small window whose size n_{small} varies from 1 to 2^{22}, shown on the x-axis. The workload consists of out-of-order data items where the out-of-order distance d is half of the small window size, i.e., $d = n_{\text{small}}/2$. The _twin experiments maintain two separate trees, one for each window size. The _range experiments maintain a single tree, using a standard query for the big window and a range query for the small window.

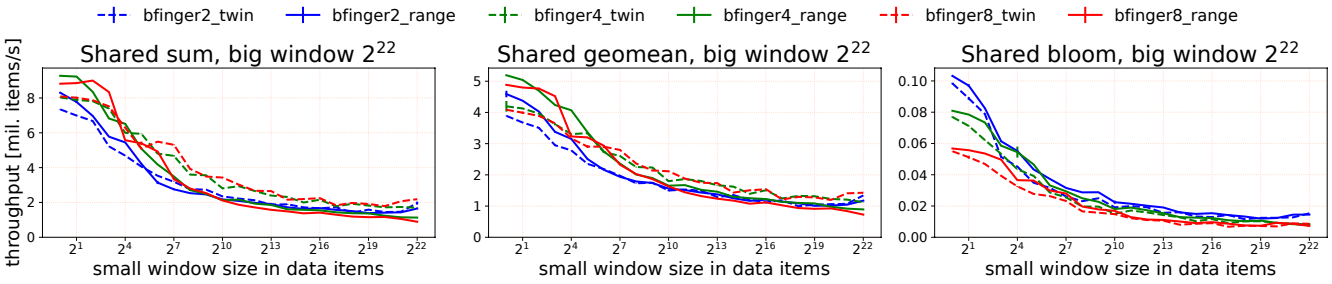

Figure 14: Window sharing experiments. Out-of-order distance also varies as $n/2$ where $n = n_{\text{small}}$ is the small window size.

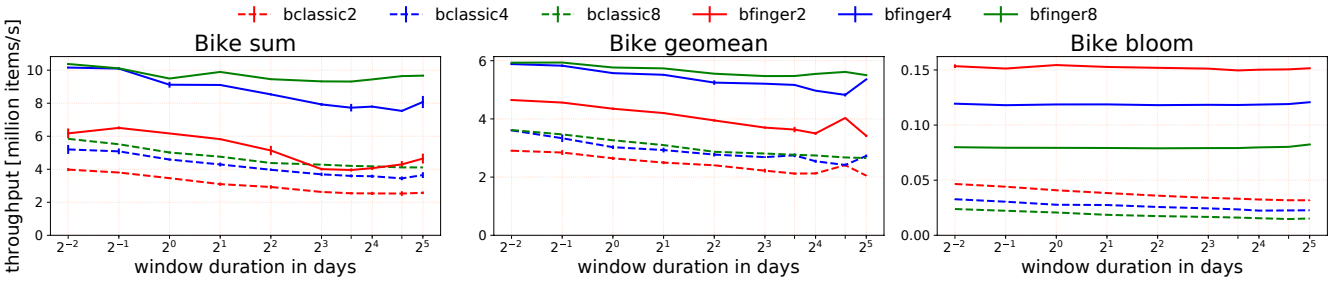

Figure 15: NYC Citi Bike data for August, 2018 through December, 2018.

Our experiment performs out-of-order insert and in-order evict, so insert costs $O(\log d)$ and evict costs $O(1)$. Hence, on average, each round of the _range experiment costs $O(\log d)$ for insert, $O(1)$ for evict, and $O(1) + O(\log n_{\text{small}})$ for query on the big window and the small window. On average, each round of the _twin experiment costs $2 \cdot O(\log d)$ for insert, $2 \cdot O(1)$ for evict, and $2 \cdot O(1)$ for query on the big and small window. Since we chose $d = n_{\text{small}}/2$, this works out to a total of $O(\log d)$ per round in both the _range and the _twin experiments. There is no fundamental reason why window sharing is slightly more expensive in practice. A more optimized code path might make range queries slightly less expensive, but we would still expect them to remain in the same ballpark.

By picking $d = n_{\text{small}}/2$, our experiments demonstrate the case where window sharing is the most likely to outperform the twin experiment. We could have increased the number of shared windows to the point where maintaining multiple non-shared windows performed worse because of the memory hierarchy, but that is the same benefit as reduced space usage. We conclude that the primary benefits of window sharing in this context are reduced space usage and the ability to query against arbitrarily-sized windows on the fly.

5.2 Real data in C++

Our real data experiments, Figure 15, use the NYC Citi Bike data [1] for two purposes: to show that our techniques work well with real out-of-order data and to showcase time-based windows. We use data from August 2018 to December 2018, for a total of about 8 million events. Each event includes trip duration, start and stop time, and start and stop location. We use start time as the event timestamp and consider events with earlier start time than any prior event to be out-of-order. The experimental environment is the same as in Section 5.1 except that it uses time-based windows. We vary the window size from 1/4 of a day to 32 days. We calculate the sum and geomean over trip duration and bloom over start location.

The real data experiments mirror the trends with synthetic data: the finger B-trees consistently outperform their classic counterparts and lower arity trees perform better with more expensive operators. The characteristics of the real data experiments are subranges within the spectrum explored with synthetic data: the actual size of the window ranges from about 11,000 elements for a time window of 1/4 of a day up to about 991,000 elements for 32 days.

In this mostly in-order dataset, out-of-order arrivals are generally mild and sporadic, but there are bursts of severely out-of-order items, concentrated in about two weeks in November. The mean out-of-order distance is $d = 56.47$ (≈ 85.9 seconds). However, up to 99% of events have $d \leq 9$ (≈ 149 seconds). The severely out-of-order bursts show up in the last 0.01%, with $d \geq 150,000$. The most severe has $d \approx 1$ million (17.7 days late).

5.3 Synthetic data in Apache Flink (Java)

How does FiBA perform relative to the state-of-the-art open-source counterparts? To answer this question as well as to understand FiBA's performance characteristics in a different environment, we reimplemented both the classic augmented B-tree and FiBA variants in Java inside Apache Flink [14]. Apache Flink was chosen because at the time of writing, it is one of the most popular open-source streaming platforms and has been the testing ground for many research projects. Our Java implementation observes idiomatic Java patterns but is otherwise the same as the C++ version. All Flink-based experiments were run on a 2-core virtual machine with Intel Xeon Platinum 8168 CPU at 2.70GHz, running Ubuntu 18.04.2 LTS with a 4.15 kernel. We compiled and ran all experiments with 64-bit OpenJDK 1.8.0_191, using Apache Flink version 1.7.1.

5.3.1 Distance-varying and FIFO. We repeated the distance-varying and FIFO experiments using as baseline Flink's built-in sliding-window aggregation (`.aggregate(<AggregateFunction>)`). The distance-varying experiment, Figure 16, uses window size $2^{13} = 8,192$ items. Though smaller than before, it is enough to study the behavior of all the algorithms without choking the baseline. FiBA and the classic augmented B-tree perform as seen previously. The throughput of Flink's built-in algorithm remains constant independent of the out-of-order distance; however, it is orders of magnitude slower than the other algorithms due to asymptotical differences.

The FIFO experiment in Figure 17 exhibits the same general trends as before, except that in this environment, the FiBA algorithms (bfinger4 and bfinger8, both $O(1)$ time for FIFO input) outperform TwoStacks (a specialized $O(1)$-time algorithm for FIFO), reversing

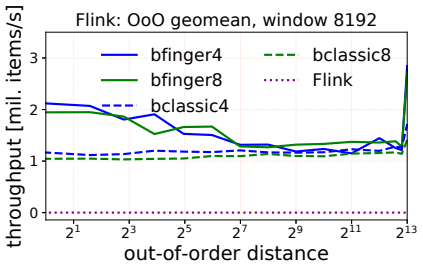

Figure 16: Out-of-order experiments on Flink

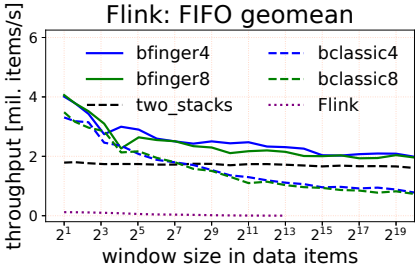

Figure 17: FIFO experiments on Flink

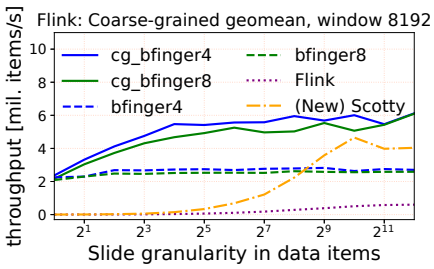

Figure 18: Coarse-grained sliding experiments

the ranking in the C++ experiments. The throughput of Flink's built-in algorithm decreases linearly with the window size and is never competitive. We stopped the Flink experiment at window size $n = 2^{13} = 8,192$, after which point each run became too expensive.

5.3.2 Coarse-grained Window. Coarse-grained windows intelligently combine items together to reduce the effective window size. The coarse-grained window experiment, Figure 18, studies how throughput (y-axis) changes with slide granularity (x-axis: how often queries are requested). The window size is $2^{13} = 8,192$, and the workload is FIFO. We present two variations on FiBA: vanilla bfinger is the standard FiBA algorithm except that queries are only made at the specified granularity, whereas cg_bfinger (coarse-grained bfinger) uses FiBA together with slicing, so items that will be evicted together are combined into one. This helps reduce the effective window size. We also include (New) Scotty, a recent work by Traub et al. [35], which improved upon Scotty [34]. The numbers reported are from their flink-connector v.0.3 code on GitHub. Flink's built-in aggregation, though never competitive, is included for reference and shows throughput improving linearly with the slide granularity.

As expected, vanilla FiBA algorithms see practically no improvement as the slide granularity increases: although queries are less frequent, the dominant cost stems from insert/evict operations, which remain the same. But Scotty's throughput improves as the window becomes coarser, ultimately outperforming vanilla FiBA (bfinger4, bfinger8) for coarse windows. However, FiBA with coarse-grained sliding (cg_bfinger) not only has the best throughput for the whole range of granularities but also exhibits performance improvement with coarser sliding granularity. This may seem counterintuitive as FiBA is already an $O(1)$-time algorithm; however, because coarse-grained sliding combines items, `insert` creates a new entry less often and `evict` occurs less frequently—hence, less total work overall.

6. RELATED WORK

Out-of-Order Stream Processing. Processing out-of-order (OoO) streams is a popular research topic with a variety of approaches. But there are surprisingly few incremental algorithms for OoO stream processing. Truviso [25] handles stream data sources that are out-of-order with respect to each other but where input values are in-order with respect to the stream they arrive on. The algorithm runs separate stream queries on each source followed by consolidation. In contrast, FiBA allows each individual stream input value to have its own independent OoO behavior. Chandramouli et al. [17] describe how to perform pattern matching on out-of-order streams but do not tackle sliding window aggregation. Finally, the Reactive Aggregator [33] performs incremental sliding-window aggregation and can handle OoO evict in $O(\log n)$ time. In contrast, FiBA can handle both OoO insert and OoO evict, and takes sub-$O(\log n)$ time.

One approach to OoO streaming is *buffering*: hold input stream values in a buffer until it is safe to release them to the rest of the pipeline [31]. Buffering has the advantage of not requiring out-of-order handling in the query since the query only sees in-order data. Unfortunately, buffering increases latency (since values endure non-zero delay) and reduces quality (since bounded buffer sizes lead to outputs computed on incomplete data). One can reduce the delay by optimistically performing computation over transactional memory [13] and performing commits in-order. Finally, one can tune the trade-off between quality and latency by adaptively adjusting buffer sizes [23]—with further optimization possible, e.g., via incremental reordering [16]. In contrast, FiBA can handle arbitrary lateness without sacrificing quality nor significant latency.

Another approach to OoO streaming is *retraction*: report outputs quickly but revise them if they are affected by late-arriving inputs. At any point, results are accurate with respect to stream input values that have arrived so far. An early streaming system that embraced this approach was Borealis [2]. Spark Streaming externalizes state from operators and handles stragglers like failures, invalidating parts of the query [37]. Pure retraction requires OoO algorithms such as OoO sliding window aggregation, but the retraction literature does not show how to do that efficiently. Our paper is complementary, describing an efficient OoO sliding window aggregation algorithm that could be used with systems like Borealis or Spark Streaming.

Using a *low watermark* (lwm) is an approach to OoO streaming that combines buffering with retraction. The lwm approach allows OoO values to flow through the query but limits state requirements at individual operators by limiting the OoO distance. CEDR proposed 8 timestamp-like fields to support a spectrum of blocking, buffering, and retraction [9]. Li et al. [28] formalized the notion of a lwm based on the related notion of punctuation [36]. StreamInsight, which was inspired by CEDR, offered a state-management interface to operator developers that could be used for sliding-window aggregation. Subsequently, MillWheel [5], Flink [14], and Beam [6] also adopted the lwm concept. The lwm leaves it to the operator developer to handle OoO values. Our FiBA algorithm is an OoO aggregation operator that could be used with systems like the ones listed above.

Sliding Window Aggregation with Sharing. All of the following papers focus on sharing over streams with the same aggregation operator, e.g., monoid $(S, \otimes, \mathbf{1})$. The Scotty algorithm supports sliding-window aggregation over out-of-order streams while sharing windows with both different sizes and slice granularities [34, 35]. For instance, Scotty might share a window of size 60 minutes and granularity 3 minutes with a session window whose gap timeout is set to 5 minutes. When a tuple arrives out-of-order, older slices may need to be updated, fused, or created. Scotty relies upon an aggregate store (e.g., based on a balanced tree) to maintain slice aggregates. FiBA could serve as a more efficient aggregate store for Scotty, thus combining the benefits of Scotty's stream slicing with asymptotically faster final aggregation.

Other prior work on window sharing requires in-order streams. The B-Int algorithm uses base intervals to share windows with different sizes [8]. Krishnamurthi et al. show how to share windows that

differ not just in size but also in granularity [26]. Cutty windows [15] extend the Reactive Aggregator [33] to share windows with different sizes and granularities. The FlatFIT algorithm performs sliding window aggregation in amortized constant time and supports window sharing [29]. Finally, SlickDeque focuses on the scenario where $x \otimes y$ always returns strictly either x or y, and offers window sharing for that scenario with a time complexity of $O(1)$ in the best case and $O(n)$ in the worst case [30]. In contrast to the above work, FiBA combines window sharing with out-of-order processing.

Finger Trees. Our FiBA algorithm uses techniques from the literature on finger trees, combining and extending them to work with sliding window aggregation. Guibas et al. [19] introduced finger trees in 1977. A *finger* can be viewed as a pointer to some position in a tree that makes tree operations (usually search, insert, or evict) near that position less expensive. Guibas et al. used fingers on B-trees, but without aggregation. Huddleston and Mehlhorn [21] offer a proof that the amortized cost of insertion or deletion at distance d from a finger is $O(\log d)$. Our analysis of tree maintenance cost is inspired by Huddleston and Mehlhorn, but simplified and addressing a different data organization: we support storing values in internal nodes, whereas Huddleston and Mehlhorn's trees store values only in leaves. Finally, Hinze and Paterson [20] present purely functional finger trees with amortized time complexity $O(1)$ at distance 1 from a finger. They describe caching a monoid-based measure at tree nodes, but this cannot be directly used for sliding-window aggregation (SWAG). Our paper is the first to use finger trees for OoO SWAG.

7. CONCLUSION

FiBA, presented in this paper, is a novel algorithm for sliding-window aggregation over out-of-order streams. Built on specially-designed finger B-trees augmented with position-aware aggregates, it works with any associative aggregation operator, does not restrict the kinds of out-of-order behavior, and supports window sharing. FiBA outperforms the prior state-of-the-art and has optimal time complexity, matching the lower bound derived in this paper.

8. APPENDIX

Time Lower Bound. For a permutation π on a set X, denote by π_i, $i = 1, \ldots, |X|$, the i-th element of the permutation. Let $\delta_i(\pi)$ be the number of elements among $\pi_1, \pi_2, \ldots, \pi_{i-1}$ that are greater than π_i—i.e., $\delta_i(\pi) = |\{j < i \mid \pi_j > \pi_i\}|$. This coincides with our out-of-order distance: if elements with timestamps $\pi_1, \pi_2, \ldots$ are inserted into OoO SWAG in that order, π_i has out-of-order distance $\delta_i(\pi)$. For an ordered set X and $d \geq 0$, let $\mathcal{G}_d(X)$ denote the set of permutations π on X such that $\max_i \delta_i(\pi) \leq d$—i.e., every element is out of order by at most d.We begin by bounding the size of such a permutation set.

Lemma 8. *For an ordered set X and $d \leq |X|$,*

$$|\mathcal{G}_d(X)| = d!(d+1)^{|X|-d}.$$

Proof. The base case is $|\mathcal{G}_0(\emptyset)| = 1$—the empty permutation. For non-empty X, let $x_0 = \min X$ be the smallest element in X. Then, every $\pi \in \mathcal{G}_d(X)$ can be obtained by inserting x_0 into one of the first $\min(|X|, d+1)$ indices of a suitable $\pi' \in \mathcal{G}_d(X \setminus \{x_0\})$. In particular, each $\pi' \in \mathcal{G}_d(X \setminus \{x_0\})$ gives rise to exactly $\min(|X|, d+1)$ unique permutations in $\mathcal{G}_d(X)$. Hence, $|\mathcal{G}_d(X)| = |\mathcal{G}_d(X \setminus \{x_0\})| \cdot \min(|X|, d+1)$. Expanding this gives

$$|\mathcal{G}_d(X)| = \prod_{k=1}^{|X|} \min(k, d+1) = \left(\prod_{k=1}^{d} k\right)\left(\prod_{k=d+1}^{|X|} d+1\right),$$

which is $d!(d+1)^{|X|-d}$, completing the proof. □

Proof of Theorem 1. Fix $X = \{1, 2, \ldots, m\}$. Let A be an OoO SWAG instantiated with the operator $x \otimes y = x$—i.e., it computes the first element of the window. Now let $\pi \in \mathcal{G}_d(X)$ be given. We will sort π using A. First, insert m elements $\left[\begin{smallmatrix}\pi_1\\ \pi_1\end{smallmatrix}\right], \left[\begin{smallmatrix}\pi_2\\ \pi_2\end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix}\pi_m\\ \pi_m\end{smallmatrix}\right]$ into A. By construction, each insertion has out-of-order distance at most d. Then, query and evict m times, reminiscent of heap sort. At this point, π has been sorted using $3m$ OoO SWAG operations in total. By a standard information-theoretic argument (see, e.g., [18]), sorting a permutation in $\mathcal{G}_d(X)$ requires, in the worst case, $\Omega(\log |\mathcal{G}_d(X)|)$ time. There are two cases to consider: If $d \leq \frac{m}{2}$, we have $|\mathcal{G}_d(X)| \geq (1+d)^{m-d} \geq (1+d)^{m-m/2} = (1+d)^{m/2}$, so $\log |\mathcal{G}_d(X)| \geq \Omega(m \log(1+d))$. Otherwise, we have $m \geq d > \frac{m}{2}$ and $|\mathcal{G}_d(X)| \geq d! \geq (m/2)!$. Using Stirling's approximation, we know $\log |\mathcal{G}_d(X)| = \Omega(m \log m)$, which is $\Omega(m \log(1+d))$ since $2m \geq 1 + d$. In either case, $\log |\mathcal{G}_d(X)| \geq \Omega(m \log(1+d))$. □

Tree Rebalancing Cost. We analyze the restructuring cost:

Lemma 9. *Let $\mu \geq 2$. The amortized cost due to tree rebalancing in a B-tree with nodes of arity between* `MIN_ARITY` $= \mu$ *and* `MAX_ARITY` $= 2\mu$ *(inclusive), starting with an empty tree initially, is $O(1)$ per OoO SWAG operation.*

Proof. This proof is a specialization of the rebalancing cost lemma in [21]. We prove this lemma by showing that if each `insert` and `evict` is billed two coins, the following invariant can be maintained for every B-tree node. Let w be a node with arity a. In a tree with minimum arity μ and maximum arity 2μ, during the intermediate steps, the arity of a node always has arity between $\mu - 1$ and $2\mu + 1$ (inclusive). We maintain a coin reserve of

$$coins(w) = \begin{cases} 4 & \text{if } a = 2\mu + 1 \\ 2 & \text{if } a = 2\mu \text{ or } (a = \mu - 1 \text{ and } w \text{ is not the root}) \\ 1 & \text{if } a = \mu \text{ and } w \text{ is not the root} \\ 0 & \text{if } a < 2\mu \text{ and } (a > \mu \text{ or } w \text{ is the root}) \end{cases}$$

To `insert` or `evict`, the data structure locates a node where an entry is either added or removed. Either way, $coins(\cdot)$ of this node never changes by more than 2, so 2 coins can cover the difference. But this may trigger a chain of `split`s or `merge`s. Below, we argue that the coin reserve on each node can pay for such `split`s and `merge`s.

When `split` is called on a node w, then w has arity $2\mu + 1$, so w has a reserve of 4 coins. When w is split, it is split into two nodes ℓ and r, with one entry promoted to w`.parent`, the parent of w. Node ℓ will have arity $\mu + 1$ and node r will have arity μ. Because $\mu < \mu + 1 < 2\mu$, we have $coins(\ell) = 0$ and node ℓ needs no coin. But node r has $coins(r) = 1$, so it needs 2 coins. Moreover, now that the arity of w`.parent` is incremented, node w`.parent` may need up to 2 additional coins. Out of 4 coins w has, use 1 to pay for the split, give 1 to r, and give up to 2 to w`.parent`, refunding any excess.

When `merge` is called on a node w, it has arity $\mu - 1$ and the sibling to merge with has arity μ. Between these two nodes, we have $2 + 1 = 3$ coins in reserve. Once merged, the node has arity $\mu + \mu - 1 = 2\mu - 1$, so it needs 0 coins. As a result of merging, the parent of w loses one child, so it may potentially need 1 coin. Out of 3 coins in reserve, use 1 for the merge and give up to 1 to w`.parent`.

Finally, note that each of `heightIncrease`, `heightDecrease`, and `move` can happen at most once for each OoO SWAG update. The internal operations `heightIncrease` and `heightDecrease` are easy to account for. For `move`, when called on a node w, it must be that w has arity $\mu - 1$, and the sibling it is interacting with has arity a', where $\mu \leq a' \leq 2\mu$. So, w has 2 coins. Once moved, w has arity μ, so it needs only 1 coin, leaving 1 coin for the sibling. The sibling of w will loose one arity, so it needs at most 1 more coin (either going from arity $\mu + 1$ to μ, or μ to $\mu - 1$). This concludes the proof. □

9. REFERENCES

[1] Citi Bike System Data. https://www.citibikenyc.com/system-data, 2019. Retrieved April, 2019.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[3] adamax. 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 Oct., 2018.

[4] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *Symposium on Principles of Database Systems (PODS)*, pages 23–34, 2012.

[5] 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. *PVLDB*, 6(11):1033–1044, 2013.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[7] M. H. Ali, C. Gerea, B. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft CEP server and online behavioral targeting. *PVLDB*, 2(2):1558–1561, 2009.

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

[9] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Conference on Innovative Data Systems Research (CIDR)*, pages 363–373, 2007.

[10] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM (CACM)*, 13(7):422–426, 1970.

[12] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online MapReduce computations. *PVLDB*, 7(13):1441–1451, 2014.

[13] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 265–275, 2008.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[15] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *Conference on Information and Knowledge Management (CIKM)*, pages 1201–1210, 2016.

[16] B. Chandramouli, J. Goldstein, and Y. Li. Impatience is a virtue: Revisiting disorder in high-performance log analytics. In *International Conference on Data Engineering (ICDE)*, pages 677–688, 2018.

[17] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, 3(1):220–231, 2010.

[18] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[19] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Symposium on the Theory of Computing (STOC)*, pages 49–60, 1977.

[20] R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming (JFP)*, 16(2):197–217, 2006.

[21] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[22] M. Izbicki. Algebraic classifiers: A generic approach to fast cross-validation, online training, and parallel training. In *International Conference on Machine Learning (ICML)*, pages 648–656, 2013.

[23] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Quality-driven processing of sliding window aggregates over out-of-order data streams. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 68–79, 2015.

[24] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *PVLDB*, 7(10):797–808, 2014.

[25] 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.

[26] 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.

[27] 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.

[28] 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. *PVLDB*, 1(1):274–288, 2008.

[29] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. FlatFIT: Accelerated incremental sliding-window aggregation for real-time analytics. In *Conference on Scientific and Statistical Database Management (SSDBM)*, pages 5.1–5.12, 2017.

[30] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. SlickDeque: High throughput and low latency incremental sliding-window aggregation. In *Conference on Extending Database Technology (EDBT)*, pages 397–408, 2018.

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

[32] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 66–77, 2017.

[33] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[34] J. Traub, P. Grulich, A. R. Cuellar, S. Breš, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: Efficient window aggregation for out-of-order stream processing. In *Poster at the International Conference on Data Engineering (ICDE-Poster)*, 2018.

[35] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *Conference on Extending Database Technology (EDBT)*, 2019.

[36] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *Transations on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, 2003.

[37] 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.