Out-of-Order Sliding-Window Aggregation with Efficient Bulk Evictions and Insertions

Kanat Tangwongsan
Mahidol University International College (MUIC)

Martin Hirzel
IBM Research

Scott Schneider
Meta
a stream = a (potentially infinite) sequence of data
a **stream** = a (potentially infinite) sequence of data

processing such infinite streams often hinges on defining a **sliding-window** on it
a stream = a (potentially infinite) sequence of data

processing such infinite streams often hinges on defining a sliding-window on it

as new data arrives...
it is inserted in the back...
... oldest data is evicted
a stream = a (potentially infinite) sequence of data

Sliding-Window Aggregation

Combine data items in time-order using a binary operator.
E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.
**Context & Motivation**

A *stream* = a (potentially infinite) sequence of data

- **Sliding-Window Aggregation**
  Combine data items in time-order using a binary operator. E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.

- **Out-of-order Streams**
  Data items are timestamped. The newest arrivals may be older than the most recent previous arrivals. E.g., clock skews across IoT devices.
Context & Motivation

A stream = a (potentially infinite) sequence of data

Processing such infinite streams often hinges on defining a sliding-window on it as new data arrives... it is inserted in the back... ... oldest data is evicted

Sliding-Window Aggregation
Combine data items in time-order using a binary operator. E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.

Out-of-order Streams
Data items are timestamped. The newest arrivals may be older than the most recent previous arrivals. E.g., clock skews across IoT devices.

Bulk Arrivals/Departures
Multiple data items enter/leave the window at once. E.g., catching up after an outage.
## Prior and Related Work

<table>
<thead>
<tr>
<th>AMTA</th>
<th>Sliding-Window Aggregation</th>
<th>Out-of-order Support</th>
<th>Bulk Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Villalba-Berral-Carrera, TPDS’19</td>
<td>✓</td>
<td>❌</td>
<td>Only bulk eviction, taking $O(\log n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DABA Lite</th>
<th>Sliding-Window Aggregation</th>
<th>Out-of-order Support</th>
<th>Bulk Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.-Hirzel-Schneider, VLDBJ’21</td>
<td>✓</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FiBA</th>
<th>Sliding-Window Aggregation</th>
<th>Out-of-order Support</th>
<th>Bulk Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.-Hirzel-Schneider, VLDB’19</td>
<td>✓</td>
<td>✓</td>
<td>❌</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Structure Papers</th>
<th>Sliding-Window Aggregation</th>
<th>Out-of-order Support</th>
<th>Bulk Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown-Tarjan’79, Kaplan-Tarjan’95, Hinze-Paterson’06</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

**Other work and techniques:** Scotty, CPIX, ChronicleDB, Hammer Slide, LightSaber, FlatFIT
This Paper: Efficient Bulk Evictions and Insertions

FiBA
[T.-Hirzel-Schneider, VLDB’19]

- Sliding-Window Aggregation: ✔ Amortized $O(\log d)$
- Out-of-order Support: ✔
- Bulk Handling: ❌
This Paper: Efficient **Bulk** Evictions and Insertions

<table>
<thead>
<tr>
<th>Sliding-Window Aggregation</th>
<th>Out-of-order Support</th>
<th>Bulk Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Amortized $O(\log d)$</td>
<td>✓</td>
<td>❌</td>
</tr>
</tbody>
</table>

**FiBA**
[T.-Hirzel-Schneider, VLDB’19]

**Make FiBA natively support bulk operations**

- `bulkInsert(B)` - Add a bulk of ordered data to the window
- `bulkEvict(t)` - Remove all items with timestamps $\leq t$
- Keep query (whole window + range) the same
This Paper: Efficient Bulk Evictions and Insertions

Theorem [This Paper]:
- bulkEvict in amortized $O(\log m)$ time
- bulkInsert in amortized $O(m \log \frac{d}{m})$ time
- query in worst-case $O(1)$

where $n =$ window size, $m =$ the bulk size and $d =$ out-of-order distance = # of data items in the window that overlap with the bulk

Make FiBA natively support bulk operations

- bulkInsert($B$) - Add a bulk of ordered data to the window
- bulkEvict($t$) - Remove all items with timestamps $\leq t$
- Keep query (whole window + range) the same
This work builds on FiBA

Finger B-Tree Aggregator

[T.-Hirzel-Schneider, VLDB’19]

Timestamp-ordered B-Tree keeping data in internal + leaf nodes

Left and right fingers for faster searching

Position-aware partial aggregates
bulkEvict, intuitively...

To support bulkEvict(t)...
bulkEvict, intuitively...

To support bulkEvict(t)...

1. **Boundary search from a finger as if looking for** $t$
   
   **Goal:** List every node on the discard-keep boundary and its right neighbor
bulkEvict, intuitively...

To support $\text{bulkEvict}(t)$...

1. **Boundary search from a finger as if looking for $t$**
   - **Goal:** List every node on the discard-keep boundary and its right neighbor

2. **A pass up along the boundary towards the root**
   - **Goal:** Disconnect nodes to discard and repair the affected nodes towards the root
bulkEvict, intuitively...

To support bulkEvict(t)...

1. **Boundary search from a finger as if looking for** $t$
   **Goal:** List every node on the discard-keep boundary and its right neighbor

2. **A pass up along the boundary towards the root**
   **Goal:** Disconnect nodes to discard and repair the affected nodes towards the root

3. **A pass down to clean up nodes on the updated spine(s)**
   **Goal:** Fix the spine(s) and repair spine aggregates
bulkInsert, intuitively...

To support bulkInsert(B)...

[Diagram of a tree structure representing bulk Insert B]
bulkInsert, intuitively...

To support bulkInsert(B)...

1. **Search for insertion sites, starting with the oldest entry in the batch**

   **Goal:** Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.
bulkInsert, intuitively...

To support `bulkInsert(B)`...

1. **Search for insertion sites, starting with the oldest entry in the batch**
   
   **Goal:** Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.
bulkInsert, intuitively...

To support bulkInsert(B)...

1. **Search for insertion sites, starting with the oldest entry in the batch**
   **Goal:** Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

2. **Level-by-level pass up to add in new entries and split overflowing nodes**
   **Goal:** No more overflowing nodes and all new entries incorporated
bulkInsert, intuitively...

To support bulkInsert(B)...

1. **Search for insertion sites, starting with the oldest entry in the batch**
   - **Goal:** Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

2. **Level-by-level pass up to add in new entries and split overflowing nodes**
   - **Goal:** No more overflowing nodes and all new entries incorporated

3. **A pass down to clean up nodes on the updated spine(s)**
   - **Goal:** Fix the spine(s) and repair spine aggregates
bulkEvict(t) is about to remove $m$ data items...
bulkEvict(t) is about to remove $m$ data items…
Implementation Consideration

discard

bulkEvict(t) is about to remove \( m \) data items…
Implementation Consideration

bulkEvict(t) is about to remove $m$ data items...

**Concern:** When \( \text{bulkEvict} \) removes \( m \) data items, it needs to discard \( O(m) \) nodes but can’t afford to eagerly free them.
Implementation Consideration

bulkEvict(t) is about to remove $m$ data items...

**Concern:** When bulkEvict removes $m$ data items, it needs to discard $O(m)$ nodes but can’t afford to eagerly free them.

**Solution:** Only eagerly free those on boundary (same as the search cost) and store their children in a **deferred free list** for future (re)use/disposal.
**Experimental Analysis**

1. How does *native bulkEvict* alter the latency profile?

2. How does *native bulkInsert* alter the latency profile?

3. Does it matter on real-world data with wildly-fluctuating window sizes and out-of-order levels?

---

**Environment Details**

- **Lang/Compiler**: C++, g++ 9.4.0 using -O3
- **OS**: Ubuntu Linux 20.04.5, Kernel 5.4.0
- **Machine**: Intel Xeon 4310 @ 2.1Ghz (exp. run single-threaded)
How does native bulkEvict alter the latency profile?

Window size $n = 4M$, bulk size $m = 1,024$

![Diagram showing performance](image_url)
How does *native* bulkEvict alter the latency profile?

Window size $n = 4M$, bulk size $m = 1,024$

Native bulkEvict translates to improved throughput as well.
How does native bulkInsert alter the latency profile?

Window size $n = 4\text{M}$, bulk size $m = 1,024$, still using \textit{geomean} faster

FIFO (in-order)

Out-of-order $d=1,024$
How does *native* bulkInsert alter the latency profile?

Window size $n = 4M$, bulk size $m = 1,024$, still using **geomean** faster

FIFO (in-order)

Out-of-order $d=1,024$

.. translates to improved throughput as well
Does it matter on real-world data with wildly-fluctuating window sizes and out-of-order levels?

**Window Size @ 1 day**

NYC Citi Bike Data (Aug - Dec 2018)

- **n** (left): Histogram of (left) citi bike instantaneous window sizes of hundreds or thousands of entries. The figure shows this for the NYC Citi Bike data.
- **m** (middle): Histogram of out-of-order distances for a time-based window of 1 day. Depending on whether that operation to give inconsistent with our results, real applications tend to use time-based windows (causal).
- **d** (right): Histogram of out-of-order distances of in the tens of thousands.

**Figure 15**: Histograms of (left) citi bike instantaneous window sizes of hundreds or thousands of entries. The figure shows this for the NYC Citi Bike data.

**Figure 16**: Throughput, citi bike, varying window size.

**Figure 17**: Throughput, Flink, bulk evict only, window size.

We ran experiments that repeatedly perform several single inserts and single inserts are uneven. Similarly, depending on the timestamp of the newest entry, it can cause a varying number operations to give inconsistent results, real applications tend to use time-based windows (causal).

This paper describes algorithms for bulk insertions and evictions. Hence, besides handling bulk operations, our algorithms also handle that case. Our algorithms are carefully crafted to yield the same algorithmic complexity as the best prior work for the non-bulk operations to give inconsistent results, real applications tend to use time-based windows (causal).

In other words, all three variables vary productively, so we report a comparison at $n = 3 	imes 10^4$ and $m = 3 	imes 10^3$, i.e., the number of records skipped over by insertions. Faster than Flink. Using bulk evictions further widens that gap.

Figure 15 shows the throughput results for the Citi Bike dataset, in the tens of thousands. The figure also shows a histogram of window sizes.

**Figure 17**: Throughput, Flink, bulk evict only, window size.

$F_{\text{flink}} = \frac{\text{bulkEvict}}{\text{in-order data}}$ (Aug–Dec 2018). The $F_{\text{flink}}$ was prohibitively slow, so we report a comparison at $n = 8,192$, varying bulk size.

Some graphs showing throughput results for the NYC Citi Bike dataset.

Faster

Future work could explore trade-offs between in-order and out-of-order data.
Bulk FiBA: Take-Away Points

- **Efficient bulk** eviction/insertion (asymptotically better)

- Retain FiBA’s efficient tuple-at-a time + queries

- Plenty more in the paper: proof(s), Flink experiments, other benchmarks, n = 1Billion, etc.

- Code is public on GitHub:
  https://github.com/IBM/sliding-window-aggregators