Tutorial: Sliding-Window Aggregation Algorithms

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Scott Schneider
IBM Research
scott.a.s@us.ibm.com

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

Abstract

Stream processing is important for analyzing continuous streams of data in real time. Sliding-window aggregation is both needed for many streaming applications and surprisingly hard to do efficiently. Picking the wrong aggregation algorithm causes poor performance, and knowledge of the right algorithms and when to use them is scarce. This paper was written to accompany a tutorial, but can be read as a stand-alone survey that aims to better educate the community about fast sliding-window aggregation algorithms for a variety of common aggregation operations and window types.

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

Keywords Survey, incremental algorithms, windows, aggregation

ACM Reference format:
Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Tutorial: Sliding-Window Aggregation Algorithms. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 4 pages.
DOI: 10.1145/3093742.3095107

1 Introduction

Aggregation is a common important feature in streaming applications. Such applications often need an aggregated summary of the most recent data in a stream, which is deemed the most relevant. The most recent data in a conceptually infinite stream is captured by a sliding window, which has an intuitive meaning to the user and a clear specification for the platform developer. Unfortunately, it is nontrivial to perform sliding-window aggregation efficiently. Naïve approaches waste time and/or resources. A poorly chosen algorithm can cause high latencies and bloated memory consumption, leading to losses, missed opportunities, and quality-of-service violations. Furthermore, in practice, users may find that their streaming platform does not support a particular aggregation operation or window kind, or if it does, may not use the most efficient algorithm.

This paper aims to provide a concise but thorough exploration of sliding-window aggregation algorithms, from theoretical and practical perspectives. The goals are:

- To enable *practitioners* to hand-implement cases not handled by their streaming platform of choice.
- To enable *streaming platform engineers* to extend the set of supported cases and use the best algorithms.
- To enable *researchers* to notice literature gaps and to stand on the shoulder of giants when filling them.

The discussion in this paper makes a distinction between the aggregation algorithm, the aggregation operation, and the window. An algorithm, such as Subtract-on-evict, can be composed with an operation, such as *avg*, and a window, such as a 10-minute window with 1-second granularity. Most algorithms address certain families of operations and windows, but have limitations that make them inapplicable to other combinations. Each algorithm comes with its own data structures for storing the window contents, as well as incremental aggregation state. Section 2 surveys operations and Section 3 discusses windows. The bulk of the paper is the survey of algorithms in Section 4.

To make the discussion more precise, we define an abstract data type for sliding window aggregation (SWAG) as follows:

- *insert*(v, t) inserts value v into the window at time t. In the first-in first-out (FIFO) case, t is younger than all times currently in the window.
- *evict*(t) removes all values whose time is t or older from the window. In the FIFO case, t is the oldest time in the window.
- *query*(t_{start}, t_{end}, x) returns the aggregation of the window between t_{start} and t_{end} with argument x. The time arguments enable window sharing, and the extra argument x can be used in case the aggregation result is itself parametric.

Not all SWAG implementations support non-FIFO windows or window sharing, and most queries are non-parametric. In those cases, we simplify the operations to *insert*(v), *evict*$()$, and *query*$()$.

SWAGs are evaluated on their throughput, latency, and memory footprint. The rest of the paper will mainly discuss these metrics via theoretical algorithmic complexity: throughput and latency are driven by average-case and worst-case time complexity, and footprint depends on space complexity. We made a conscious decision to keep the presentation independent of any particular streaming platform by tackling the subject matter at an algorithmic level. That said, actual performance in practice also depends on other factors. For instance, footprint is sometimes less important than the active working set, which determines cache misses and thus indirectly influences latency and throughput.

2 Aggregation Operations

We want to apply aggregation operations incrementally, as data items are inserted and evicted from a sliding window. To that end, it is useful to break down each operation into the three functions *lift*, *combine*, and *lower* as follows:

- *lift* turns a stream data item into an intermediate result.
  E.g.: *avg.lift*$(d) = \langle d, 1\rangle$.
- *combine* merges pairs of intermediate results.
  E.g.: *avg.combine*$(\langle s_1, n_1\rangle, \langle s_2, n_2\rangle) = \langle s_1 + s_2, n_1 + n_2\rangle$.
  The choice to *combine* two intermediate results rather than one intermediate result and one input stream data item is essential for some aggregation algorithms. We agree with Guy Steele's assessment "foldl and foldr considered slightly harmful" [18].
- *lower* turns an intermediate result into a final result.
  E.g.: *avg.lower*$(\langle s, n\rangle) = s/n$.

	invertible combine	associative combine	commutative combine	size-preserving combine	unary lower
• **sum-like**: sum, count, average, standard deviation, ...	✓	✓	✓	✓	✓
• **collect-like**: collect list, concatenate strings, i^{th}-youngest, ...	✓	✓	×	×	?
• **median-like**: median, percentile, i^{th}-smallest, ...	✓	✓	✓	×	?
• **max-like**: max, min, argMax, argMin, maxCount, ...	×	✓	?	✓	✓
• **sketch-like**: Bloom filter [3], CountMin [5], HyperLogLog [6]	×	✓	✓	✓	×

Table 1. Aggregation operations. Checkmarks (✓), crosses (×), and question marks (?) indicate a property is true for all, false for all, or false for some of a given group of like operations, respectively.

	Complexity		Applicability
	Time	Space	restrictions
• Recalculate from scratch	worst-case $O(n)$	$O(n)$	no restrictions
• Subtract on evict	worst-case $O(1)$	$O(n)$	sum-like or collect-like
• Order statistics tree [9]	worst-case $O(\log n)$	$O(n)$	median-like
• Reactive Aggregator [20]	average $O(\log n)$	$O(n)$	size-preserving, assoc.
• Two-stacks [1]	average $O(1)$	$O(n)$	size-preserving, assoc., FIFO
• DABA [19]	worst-case $O(1)$	$O(n)$	size-preserving, assoc., FIFO
• B-Int [2]	shared $O(\log n_{\max})$	$O(n_{\max})$	size-preserving, assoc., FIFO

Table 2. Aggregation algorithms.

In this example, `lower` is unary, but in some cases, it takes additional arguments, such as i for i^{th}-smallest, or an element identifier for a Bloom filter membership test.

Table 1 lists aggregation operations, characterizes them with properties, and groups them into categories. For example, *avg* belongs to the sum-like category, for which all properties we consider hold.

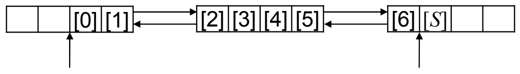
Let us define $\oplus$ as a binary operator version of the *combine* function. An aggregation function is *invertible* if there exists some function $\ominus$ such that $(x \oplus y) \ominus y = x$ for all x and y. SWAGs can take advantage of invertible aggregation functions by implementing deletion as an undo; just apply $\ominus$ to the accumulated state. A function is *associative* if $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ for all x, y, and z. SWAGs can take advantage of associativity by applying $\oplus$ at arbitrary places inside the window. Without associativity, SWAGs are restricted to applying $\oplus$ only at the end, upon insertion. A function is *commutative* if $x \oplus y = y \oplus x$ for all x and y. SWAGs are able to ignore data item insertion order for commutative aggregation operators. A function is *size-preserving* if the result of $\oplus$ is always the same size in memory. Aggregation functions that are size preserving will have constant memory costs. Finally, aggregation functions with a *unary lower* do not require any additional information to apply the `lower` function to produce a result.

The set of properties we chose harmonizes properties of aggregation operations defined in prior work [8, 20]. Even though some properties are specific to only the *combine* function, we slightly abuse terminology to let them refer to the entire operation. For instance, we say the *avg* operation is invertible because the *avg.combine* function is invertible.

3 Windows

This section only discusses window interfaces and implementation techniques germane to aggregation. Windows are exhaustively cataloged [7] and formally treated [4] elsewhere.

We begin by outlining a chunked-array queue implementation for FIFO windows [19]. The main operations are *pushBack*(v) and *popFront*(), which, respectively, insert v at the back of the queue and evict the front element from the queue. Additionally, the queue provides a bidirectional iterator, which can be thought of as pointers into the queue. To support all operations in $O(1)$ time using $O(n)$ space, it maintains a doubly-linked list of chunks, each a fixed-size array of elements. This means allocation only happens when the rear chunk is full and deallocation when the front chunk becomes empty. By contrast, an implementation that keeps a single array (e.g., using C++'s STL *vector* or Java's *ArrayList*) occasionally spends linear time to grow or shrink the underlying array.



Window Optimization Techniques: When the `lift` function is deterministic, it only has to be applied to each element once when it first arrives; the lifted version is kept in the window, saving the cost of `lift` calls. Furthermore, data items that will be evicted together may be combined at insertion [11, 12]. This not only saves space but also reduces the effective window size n, thereby improving the running time. Another common technique is window partitioning, sometimes used as a means to maintain grouped-by aggregation and obtain data parallelism using fission [15]. This can be implemented by running multiple independent instances of a SWAG, one per partition, and keeping a (hash-)map that relates attributes in data items to the appropriate instance.

4 Aggregation Algorithms

This section covers each of the algorithms in Table 2. In the complexity columns, n refers to the current window size. For example, each insertion to or eviction from an order statistics tree takes at most $O(\log n)$ time. The time complexity of the B-Int algorithm is given as "shared $O(\log n_{\max})$", by which we mean that B-Int simultaneously aggregates multiple windows of different sizes using time logarithmic in the size $n_{\max}$ of the largest window. These algorithms are the best-performing approaches for a range of circumstances. The circumstances are summarized in the applicability-restrictions column, which refers to properties of aggregation operations and of windows. For example, the DABA algorithm works for any

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

Figure 1. Recalculate-from-scratch SWAG algorithm.

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

Figure 2. Subtract-on-evict SWAG algorithm.

```
1 fun insert(v)
2   vals.pushBack(v)
3   tree.insert(v.k, v)
4 fun evict()
5   v ← vals.popFront()
6   tree.remove(v.k, v)
7 fun query(i)
8   return tree.select(i)
```

Figure 3. Order statistics tree SWAG algorithm.

```
1  fun insert(v)
2    B.push(v, $\Sigma_B^{\oplus} \oplus v$)
3  fun evict()
4    if F.isEmpty() // Flip
5      while not B.isEmpty()
6        F.push(B.top().val, B.top().val $\oplus \Sigma_F^{\oplus}$)
7        B.pop()
8    F.pop()
9  fun query()
10   return $\Sigma_F^{\oplus} \oplus \Sigma_B^{\oplus}$
11 fun $\Sigma_F^{\oplus}$:  F.isEmpty() ? $\bar{0}$ : F.top().agg
12 fun $\Sigma_B^{\oplus}$:  B.isEmpty() ? $\bar{0}$ : B.top().agg
```

Figure 4. Two-stacks SWAG algorithm.

size-preserving associative aggregation operations if the window is strictly first-in-first-out (FIFO).

4.1 Recalculate from Scratch

Figure 1 shows the simplest and most general SWAG algorithm, which just aggregates the window contents with a loop on every *query* call. The $\bar{0}$ sign in Line 6 represents the identity element for $\oplus$, meaning $x \oplus \bar{0} = x = \bar{0} \oplus x$ for all x. This algorithm requires no fancy data structure. If the window is FIFO, the values `vals` can just be kept in a chunked-array queue from Section 3. Unfortunately, the approach takes $O(n)$ time, making it prohibitively expensive for large windows.

4.2 Subtract on Evict

When the aggregation operation's $\oplus$ has an inverse $\ominus$, the SWAG can maintain a current aggregation *agg* by adding and subtracting for insert and evict, as shown in Figure 2. When it is applicable, Subtract-on-evict takes $O(1)$ time and is simple to implement. Unfortunately, many aggregation operations are not invertible, and even arithmetic addition is not invertible for floating point.

4.3 Order Statistics Tree

An order statistics tree is a variant of a balanced search tree that supports an additional `select(i)` operation for finding the i^{th} smallest value in the tree.[1] It can be used to implement a SWAG for median-like aggregation operations in $O(\log n)$ time [9]. The idea is to place each stream value both in a queue and in the order statistics tree. Then, for instance, median is implemented as *query*$(\lfloor n/2 \rfloor)$, and the p^{th} percentile is *query*$(\lfloor n \cdot \frac{p}{100} \rfloor)$. Figure 3 shows the algorithm for the general case of computing the i^{th}-smallest value, where i is a runtime argument.

4.4 Reactive Aggregator

The Reactive Aggregator (RA) works for any size-preserving associative aggregation operation, even non-invertible ones, and does not require FIFO windows [20]. RA is implemented via a balanced tree ordered by time, where internal nodes hold the partial aggregations of their subtrees, and offers $O(\log n)$ performance. Instead of the conventional approach to implementing balanced trees by frequent rebalancing, RA projects the tree over a complete perfect binary tree, which it stores in a flat array. Doing so requires some extra care of how the intermediate results are stored and combined. In return, the flat-array implementation leads to higher performance than other tree-based SWAG implementations in practice, since it saves the time of rebalancing as well as the overheads of pointers and fine-grained memory allocation.

One of the advantages of RA is that it can handle non-FIFO windows as long as the aggregation operation is commutative. Aside from RA, there are also other, orthogonal approaches for handling out-of-order streams and non-FIFO windows. Krishnamurthy et al. show how to reconcile a small number of streams that drift arbitrarily far from each other by merging their pre-aggregated values [10]. Srivastava and Widom show how to reconcile stream values arriving with a large variety of small delays by placing new values in a holding buffer first [17].

4.5 Two-Stacks

When the window is FIFO, the running time can be substantially improved. The Two-stacks algorithm [19], building on prior specialized solutions [1, 16], maintains two aggregating stacks. Aggregation is easy on a stack: by storing with every element the aggregation of all data items below it, the algorithm can support stack operations in $O(1)$ time. Using an old functional programming trick, a queue (SWAG) is maintained as two stacks, front F and back B. The whole algorithm is simple, as shown in Figure 4. As a data item arrives (`insert`), it is pushed into B and later popped from F when evicted (`evict`). Transfer from B to F happens when `evict` needs to pop an item from F but F is empty. Each transfer empties out B and reinserts the items into F, costing $O(n)$ time. Since transfer only happens occasionally, Two-stacks achieves $O(1)$ amortized time for all SWAG operations, yielding impressive throughput in practice. But because it can occasionally spend linear time in `evict`, Two-stacks is inappropriate for latency-critical situations.

4.6 DABA

For latency-sensitive applications, the aggregation algorithm cannot afford a long pause. Building on Two-stacks, the De-Amortized Banker's Aggregator (DABA) [19] ensures that every SWAG operation takes $O(1)$ time in the worst-case, not just on average. While Two-stacks must occasionally flip the back stack (B) onto the front stack (F), costing $O(n)$ time, DABA gradually performs this operation, making some progress with every *pushBack* and *popFront*. This is accomplished by extending Okasaki's functional queue [13], which guarantees that by the time a data item is needed in F, it has already been transferred over. However, Okasaki's functional

[1]`https://en.wikipedia.org/wiki/Order_statistic_tree`

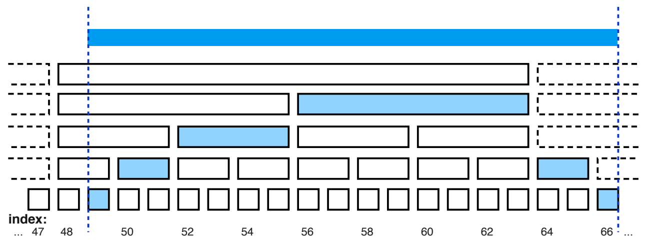


Figure 5. Base intervals between indices 47 and 66. Shaded in blue are base intervals that make up interval [49, 66].

queue depends on lazy evaluation and automatic garbage collection. DABA sidesteps these requirements by performing all the actions on a single chunked-array queue, with different moving parts maintained as pointers into the queue. This has been shown to keep the overhead small while supporting SWAG operations in worst-case $O(1)$ time as long as *combine* is a size-preserving associative operator and the window is FIFO.

4.7 B-Int

All the algorithms discussed so far were designed to answer aggregation queries about one specific window. Unlike these algorithms, B-Int (short for base intervals) [2] was designed to facilitate sharing between different windows. It does so by storing a "shared" window S that contains inside it all the windows being shared. In this manner, aggregation queries between any two endpoints can be answered as long as the endpoints lie within the shared window. But this by itself does not yield fast SWAG operations. In addition to keeping the contents of S, B-Int maintains certain preaggregated values, so that a query between the i-th data item and j-th data item within S can be answered by combining at most $O(\log|i-j|)$ preaggregated values, resulting in logarithmic running time.

These preaggregated values are derived for all base intervals that lie within S. *Base intervals* (more commonly known now as dyadic intervals) are intervals of the form $[2^\ell k, 2^\ell(k+1)-1]$ with $\ell, k \geq 0$. The parameter ℓ defines the level of a base interval. The base interval $[i, j]$ contains all data items between the i-th data item and the j-th data item in the stream. That is, whether a data item belongs in a base interval depends on the position of this data item in the stream, not its position in the window. Figure 5 shows all base intervals relevant to indices between 47 and 66. Notice how the interval [49, 66], which is not a base interval itself, is made up of a small number of base intervals (shaded in blue). To answer an aggregation query about interval [49, 66], B-Int combines the preaggregated values of these shaded intervals. When the size of S is fixed at $n_{\max}$, each level of base intervals can be stored in an array—as a circular buffer—using $O(n_{\max}/2^\ell)$ space, leading to a footprint of $O(n_{\max})$ overall. B-Int is applicable provided that *combine* is associative and size-preserving and the window is FIFO.

5 Conclusions

A simple way to speed up sliding-window aggregation (SWAG) is to select the right algorithm. Algorithm selection is easier than many other optimizations for streaming applications [14]. To choose the right SWAG algorithm, one must consider the aggregation operation, the window kind, and requirements for latency, out-of-order processing, and sharing, if any. There may not be an acceptable algorithm for all combinations of requirements. When there is such a gap in the choices for algorithms, one option is to compromise on some of the requirements. For instance, one can pick a window that is easier to handle, e.g., by coarsening the granularity by which the window slides. That said, this paper aims at spreading the knowledge of available choices, so users need not compromise unnecessarily. Of course, from the perspective of researchers, literature gaps are opportunities. For instance, improving the algorithmic complexity by inventing a new algorithm for a particular set of circumstances is intellectually rewarding.

References

[1] adamax. 2011. Re: Implement a queue in which push_rear(), pop_front() and get_min() are all constant time operations. http://stackoverflow.com/questions/4802038/. Retrieved May, 2017.

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

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

[4] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *Conference on Very Large Data Bases (VLDB)*. 232–243.

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

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

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

[8] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. 1996. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *International Conference on Data Engineering (ICDE)*. 152–159.

[9] Martin Hirzel, Rodric Rabbah, Philippe Suter, Olivier Tardieu, and Mandana Vaziri. 2016. Spreadsheets for Stream Processing with Unbounded Windows and Partitions. In *Conference on Distributed Event-Based Systems (DEBS)*. 49–60.

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

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

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

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

[14] Scott Schneider, Buğra Gedik, and Martin Hirzel. 2013. Tutorial: Stream Processing Optimizations. In *Conference on Distributed Event-Based Systems (DEBS)*. 249–258.

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

[16] Jon Skeet. 2009. Re: design a stack such that getMinimum() should be O(1). http://stackoverflow.com/questions/685060/. Retrieved May, 2017.

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

[18] Guy L. Steele, Jr. 2009. Organizing Functional Code for Parallel Execution or, Foldl and Foldr Considered Slightly Harmful. In *International Conference on Functional Programming (ICFP)*. 1–2.

[19] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *Conference on Distributed Event-Based Systems (DEBS)*.

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