General Incremental Sliding-Window Aggregation

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

Martin Hirzel Scott Schneider Kun-Lung Wu
IBM Research, Yorktown Heights, NY, USA
{hirzel,scott.a.s,klwu}@us.ibm.com

ABSTRACT

Stream processing is gaining importance as more data becomes available in the form of continuous streams and companies compete to promptly extract insights from them. In such applications, sliding-window aggregation is a central operator, and incremental aggregation helps avoid the performance penalty of re-aggregating from scratch for each window change.

This paper presents Reactive Aggregator (RA), a new framework for incremental sliding-window aggregation. RA is general in that it does not require aggregation functions to be invertible or commutative, and it does not require windows to be FIFO. We implemented RA as a drop-in replacement for the Aggregate operator of a commercial streaming engine. Given m updates on a window of size n, RA has an algorithmic complexity of $O(m + m\log(n/m))$, rivaling the best prior algorithms for any m. Furthermore, RA's implementation minimizes overheads from allocation and pointer traversals by using a single flat array.

1. INTRODUCTION

Stream processing is important in time-critical applications that deal with continuous data, so much so that several streaming platforms have been developed over the past few years [2, 9, 20, 34, 38]. As our world is increasingly instrumented, stream processing has widespread uses in telecommunications, health care, finance, retail, transportation, social media, and more. Most streaming applications involve aggregation of some form or another. For instance, a trading application may aggregate the average price over the last 1,000 trades, or a network monitoring application may track the total network traffic in the last 10 minutes.

Streaming aggregation is often performed over sliding windows. Windows are a central concept in stream processing because an application cannot store an infinite stream in its entirety. Instead, windows summarize the data in a way that is intuitive to the user, as the most recent data is typically the most relevant data. Sliding windows are so fundamental to streaming systems that papers about their semantic subtleties are widely cited [7, 21].

*Corresponding author. Part of this work was done at IBM Research.

Proceedings of the VLDB Endowment, Vol. 8, No. 7

This paper introduces *Reactive Aggregator* (RA), a framework for sliding-window aggregation. RA brings together a novel combination of algorithmic techniques in a common package that is more general than prior work and at the same time can take better advantage of incremental computation than existing solutions.

RA is *general* enough to serve as a drop-in replacement for the Aggregate operator of SPL, which is the language for InfoSphere Streams, an industrial-strength streaming engine [18]. Over the years, Streams customers have requested support for a host of custom aggregations, for instance, for time series analyses or statistical functions—and that they have good performance.

Achieving this level of generality requires addressing several cases rarely supported by prior work, most notably the ability to efficiently handle non-invertible, non-commutative aggregations and non-FIFO windows. Prior work often relies on aggregation functions to be invertible. In practice, there are several non-invertible cases, including Min, Max, First, Last, CollectDistinct, and ArgMax. Furthermore, prior work often depends on aggregation functions to be commutative. In practice, there are several non-commutative cases, including First, Last, Sum<String> (i.e., concatenation), Collect, and ArgMax. For instance, ArgMax is not invertible, and its result may not be uniquely defined. A common approach for returning a unique result is using the first maximum tuple in arrival order. Doing so yields deterministic results for high-stakes and highly-regulated domains such as finance (e.g., find the highest offer) or medical (e.g., find the most severe alarm), at the cost of being non-commutative. Finally, most prior solutions require sliding windows to be strictly FIFO. In practice, windows are often defined based on timestamp attributes that may be slightly out of order. RA can handle non-invertible, non-commutative, and non-FIFO scenarios.

To understand the performance benefits that RA offers, let n be the window size and m be the update size, i.e., the number of tuples inserted or evicted before recomputation. Prior work falls into two camps: work that optimizes for small m vs. work that optimizes for large m. Work that optimizes for small m uses various forms of balanced trees, yielding $O(\log n)$ time assuming m is a constant, such as $m = 1$ [3, 26, 36]. Work that optimizes for large m uses some form of two-step aggregation, yielding $O(m)$ time assuming m is proportional to n, such as $m = n/7$ [9, 22, 23]. RA yields $O(m + m\log(n/m))$ time, rivaling the best prior work for both small and large m. To our knowledge, no prior algorithms can both achieve this time complexity and support variable-sized windows.

RA not only has good theoretical complexity but also performs well in practice because the implementation stores its state in a single flat array, minimizing pointer traversal and allocation overheads. It also uses code generation to eliminate dynamic dispatch overheads.

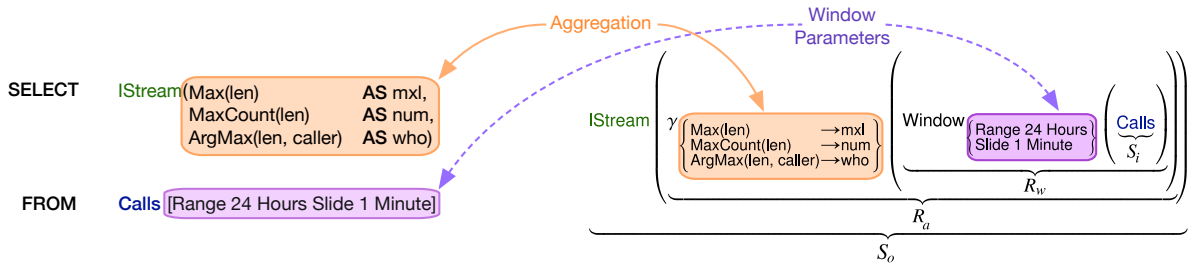

Figure 1: Example query in CQL (left) and the same query in stream-relational algebra (right).

As a framework, RA combines an efficient data structure with a simple abstraction for library developers to program aggregation operations. It requires that the aggregation operation be decomposed into lift, combine, and lower functions (to be explained); and that combine be associative (not necessarily commutative nor invertible). Furthermore, RA supports both fixed-size windows and variable-size windows (e.g., time-based windows) by resizing the data structure appropriately. We provide a resizing algorithm where the resizing overhead can be amortized and absorbed into the processing cost.

Our experiments confirm the theoretical findings and show that RA performs well in practice. For most aggregation operations, RA is never more than 10% worse than from-scratch recomputation on small windows (between 1 and 100), but is capable of delivering at least an order of magnitude higher throughput—and often much higher—on a window as small as 6K elements.

2. OVERVIEW OF OUR APPROACH

To provide context for the remainder of the paper, we formalize the supported queries, as well as discussing considerations that shaped the design and implementation.

2.1 Query Example and Semantics

This section describes the queries implemented by our reactive aggregator in terms of CQL's stream-relational algebra [4]. The reactive aggregator implements algebraic queries of the form IStream ∘ γ ∘ Window. As an example, Figure 1 shows a query that uses sliding-window aggregation to compute statistics over phone calls, transforming input stream S_i into output stream S_o.

We denote the input stream by S_i, in this example, S_i = Calls. Formally, a *stream* is a possibly infinite bag of stream elements. A *stream element* is a pair $\langle d, \tau \rangle$ of a tuple d and a timestamp τ. For the example, we assume the tuples of input stream Calls have schema {len: Float, caller: String}.

$R_w = \text{Window}_{\{\ldots\}}(S_i)$ is the window. At each time τ, a *window* converts a stream S_i into a relation $R_w(\tau)$ containing recent tuples from S_i. In the example, Range 24 Hours specifies the window size, and Slide 1 Minute specifies the slide parameter, which determines the granularity at which the window contents change. Formally, the window at time τ begins at time $\tau_b = \left\lfloor \frac{\tau - 24\text{ Hours}}{1\text{ Minute}} \right\rfloor \cdot 1\text{ Minute}$ and ends at time $\tau_e = \tau_b + 24$ Hours. The window is the bag of tuples $R_w(\tau) = \{d \,|\, \langle d, \tau' \rangle \in S_i \wedge \tau' \geqslant \tau_b \wedge \tau' \leqslant \tau_e\}$.

$R_a = \gamma_{\{\ldots\}}(R_w)$ is the aggregation. After applying a window, CQL uses standard relational algebra operators on the resulting relation. This works because a window at time τ simply returns a bag of tuples $R_w(\tau)$. The semantics of relational algebra on bags of tuples is well-understood and can be found in databases texts. Formally, the γ operator in the example computes a bag R_a with a single tuple

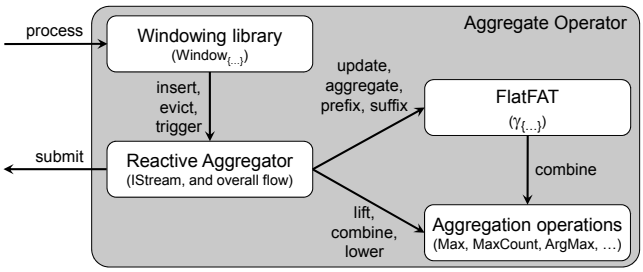

Figure 2: Overview of our approach.

that has the schema {mxl : Float, num : Int, who : String}, where

$$\text{mxl} = \max\{d.\text{len} \,|\, d \in R_w(\tau)\}$$
$$\text{num} = |\{d \,|\, d \in R_w(\tau) \wedge d.\text{len} = \text{mxl}\}|$$
$$\text{who} \in \{d.\text{caller} \,|\, d \in R_w(\tau) \wedge d.\text{len} = \text{mxl}\}$$

Note that who might not be uniquely defined. For deterministic results, the tie can be resolved to the first tuple in arrival order.

$S_o = \text{IStream}(R_a)$ turns the singleton relation R_a back into a stream. The IStream operator watches a time-varying relation $R_a(\tau)$, producing a stream element $\langle d, \tau \rangle$ whenever a tuple d is inserted into R_a at time τ.

2.2 Window Size and Update Size

While the example in Figure 1 only illustrates one particular configuration of γ and Window operators, the reactive aggregator supports other configurations. Window size can be specified by time, count, delta, or punctuation. The slide parameter can be anything from a single tuple to the entire window size, in which case the sliding window degenerates to a tumbling window. Furthermore, the window and the aggregation can be partitioned on a *key* (a set of attributes). Tuples from input stream S_i belong to the same partition if their key attributes have the same value. Each partition of the window is aggregated independently.

We define n_τ, the instantaneous window size at time τ, as the number of tuples in $R_w(\tau)$. If the user specifies a count-based window, n_τ is constant once the application reaches steady-state. For all other window size specifications (time, delta, or punctuation), n_τ can vary throughout the application run. We define m, the update size between two successive time instants τ and τ', as the number of insertions and deletions between $R_w(\tau)$ and $R_w(\tau')$. With a count-based slide parameter, m is constant; otherwise, m is variable.

2.3 Aggregate Operator

To concretely describe our solution, we explain how it works as a drop-in replacement for SPL's Aggregate operator [18]. The techniques presented in this paper, however, are platform-independent.

Figure 2 gives an overview of the Aggregate operator using our solution. In broad strokes, this works as follows. When a tuple is delivered to Aggregate, the SPL runtime calls the process function.

Operation	Types:			Functions:		
	In	Agg	Out	lift(v:In) : Agg	combine(a:Agg, b:Agg) : Agg	lower(c:Agg) : Out
Count	T	Int	Int	1	a+b	c
Sum	T	T	T	v	a+b	c
Max	T	T	T	v	a>b ? a : b	c
Min	T	T	T	v	a<b ? a : b	c
ArithmeticMean	T	{n:Int, Σ:T}	T	n=1, Σ=v	n=a.n+b.n, Σ=a.Σ+b.Σ	Σ/n
GeometricMean	T	{n:Int, Π:T}	T	n=1, Π=v	n=a.n+b.n, Π=a.Π*b.Π	$\sqrt[n]{\Pi}$
MaxCount	T	{n:Int, max:T}	Int	n=1, max=v	pick higher or add if equal	n
MinCount	T	{n:Int, min:T}	Int	n=1, min=v	pick lower or add if equal	n
SampleStdDev	T	{n:Int, Σ:T, sq:T}	T	n=1, Σ=v, sq=v^2	a.n+b.n, a.Σ+b.Σ, a.sq+b.sq	$\sqrt{\frac{1}{n-1}(\text{sq} - \Sigma^2/n)}$
PopulationStdDev	T	{n:Int, Σ:T, sq:T}	T	n=1, Σ=v, sq=v^2	a.n+b.n, a.Σ+b.Σ, a.sq+b.sq	$\sqrt{\frac{1}{n}(\text{sq} - \Sigma^2/n)}$
ArgMax	{m:T, a:T'}	{max:T, arg:T'}	T'	max=v.m, arg=v.a	pick higher or first if equal	arg
ArgMin	{m:T, a:T'}	{min:T, arg:T'}	T'	min=v.m, arg=v.a	pick lower or first if equal	arg
Collect	T	List<T>	List<T>	[v]	concat(a, b)	c

Table 1: Decomposition of a variety of aggregate operations, showing the input type In, intermediate aggregate type Agg, and output type Out; and the corresponding functions to perform the aggregation.

The operator then performs various steps (more details below) responsible for bookkeeping and generating output tuples. When it is time to produce an output tuple, the operator calls the SPL runtime to submit the tuple to the output stream.

There are several components to the Aggregate operator. The windowing library implements the $\text{Window}_{\{...\}}$ operator from Figure 1. It supports combinations of time, count, delta, sliding, tumbling, partitioning, punctuation, etc. The windowing library takes care of tracking which tuples are currently in the window, and offers a uniform interface for all the different window kinds. With the windowing library, the reactive aggregator must only handle three kinds of events: insert, evict, and trigger. Gedik describes the details of the windowing library [12].

Using our solution, the aggregation operations (Section 3) are each described by three functions (lift, combine, and lower). Users who wish to develop an aggregation operation only have to write these functions; we merely require that combine be associative.

FlatFAT (Section 4) implements the $\gamma_{\{...\}}$ operator from Figure 1. It maintains intermediate aggregates in a tree, using the combine function of the various aggregate operations at internal tree nodes. The details of FlatFAT are the core contribution of this paper.

Reactive Aggregator (Section 5) batches several insert and evict events while tracking the relevant indices in FlatFAT. When the window triggers, the reactive aggregator lifts the relevant input values based on the relevant aggregation operations, and updates FlatFAT. Then, it gets the result from the FlatFAT and calls lower from the relevant aggregation operations. Finally, it implements the IStream operator from Figure 1 by putting the results in an output tuple, and submitting that.

3. AGGREGATION OPERATIONS

As shown in Figure 2, the reactive aggregator relies on aggregation operations implemented via three functions lift, combine, and lower. This simple abstraction for building an aggregation operation shields library developers from the burden of reasoning about incrementalization and potentially tricky implementation details.

3.1 Library Developer's Perspective

Each operation works with three types: input In, partial aggregation Agg, and output Out. The three function signatures are:

- lift(v: In) : Agg computes the partial aggregation for a single-tuple subwindow;
- combine(a : Agg, b : Agg) : Agg, often rendered in the binary operator notation $a \oplus b$, transforms partial aggregations for two subwindows into the partial aggregation for the combined subwindow; and
- lower(c : Agg) : Out turns a partial aggregation for the entire window into an output.

As an example, Table 1 summarizes several aggregation operations, their types, and functions; we pick two representative operations based on the CQL example from Figure 1 to explain next.

Max is one of the simpler operations in Table 1: it uses the same type for In, Agg, and Out, and the functions lift and lower are mere identity functions. The example in Figure 1 invokes Max as follows: $\text{Max}(\text{len}) \rightarrow \text{mxl}$. The type of len is Float, and unifies with the generic type variable T in Table 1. The combine function takes partial aggregation results from two subwindows, and returns a new partial aggregation for the fused subwindow. For the case of Max, that simply means taking the larger value.

In contrast to Max, ArgMax exercises the full generality of the three types and three functions. ArgMax has two parameters, as illustrated in in the example $\text{ArgMax}(\text{len}, \text{caller}) \rightarrow \text{who}$ from Figure 1, which maximizes len while tracking the argument caller for which the maximum is reached. Unifying the concrete actual input type {len:Float, caller:String} against the generic formal input type In, which is {m:T, a:T'}, yields the substitution $[\text{T} \mapsto \text{Float}, \text{T'} \mapsto \text{String}]$. With this substitution, type Agg is {max:Float, arg:String} and type Out is String. The functions work as follows:

ArgMax.lift returns a partial aggregation for a singleton subwindow, in this example, max=len and arg=caller.

ArgMax.combine takes partial aggregations from two subwindows. It yields arg and max from the subwindow that has a higher value of max. If max is the same in both subwindows, combine resolves the ambiguity deterministically by picking arg from the first one.

ArgMax.lower takes a partial aggregation over the entire window, and produces the actual output. In the case of the running example ArgMax(len, caller), the output is simply the caller corresponding to the maximum len.

Our interface is general as demonstrated by the variety of operations in Table 1. Note that besides the generality across operations, the approach also yields generality across concrete types, by using generic types and functions. Specifically, the operations work across different input types In, and the other types Agg and Out depend on that type, as do the function signatures.

3.2 Algebraic Properties

Algebraic properties offer insights into the behavior of an aggregation operation, indicating the difficulty of incrementalization.

They set frameworks apart in terms of generality. For example, a framework that only works for invertible operations is less general than one that also works for non-invertible ones. Our reactive aggregation approach requires associativity, but not invertibility or commutativity. We remember the related definitions below.

A combine function, rendered in binary-operator notation as $\oplus$, is *associative* if $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ holds for all partial aggregation results x, y, z. Without associativity, one can only handle insertions one element at a time at the end of the window. Associativity enables breaking down the computation in flexible ways, including balanced splitting into perfect binary trees.

A combine function $\oplus$ is *invertible* if there exists a (known and reasonably cheap) function $\ominus$ such that $(x \oplus y) \ominus y = x$ holds for all partial aggregation results x, y. Invertibility enables handling deletions as inverse insertions. This is the standard approach to incremental sliding-window aggregation. In contrast, our algorithm does not require invertibility, making it more general.

A combine function $\oplus$ is *commutative* if $x \oplus y = y \oplus x$ holds for all partial aggregation results x, y. A commutative combine function makes it possible to ignore the order of the input when computing aggregation results. Our algorithm does not require commutativity.

Operation	Algebraic properties:		
	associative	invertible	commutative
Count	✓	✓	✓
Sum	✓	✓	✓
Max	✓		✓
Min	✓		✓
ArithmeticMean	✓	✓	✓
GeometricMean	✓	✓	✓
MaxCount	✓		✓
MinCount	✓		✓
SampleStdDev	✓	✓	✓
PopulationStdDev	✓	✓	✓
ArgMax	✓		
ArgMin	✓		
Collect	✓	✓	

Table 2: Algebraic properties of aggregate operations.

Table 2 shows the algebraic properties of the operations from Table 1. All the common aggregations are associative, validating our approach (and the improved algorithmic complexity it brings). Several aggregations are non-commutative, and several common aggregations are not invertible, mostly because they are not bijective. In certain cases, one can handcraft case-by-case solutions to make them bijective by introducing additional state; in contrast, we present a general solution that avoids this requirement.

4. FLAT FIXED-SIZED AGGREGATOR

The reactive aggregator framework, as Figure 2 shows, implements the relational aggregation operator $\gamma_{\{\ldots\}}$ using FlatFAT. A *flat fixed-sized aggregator* (FlatFAT) of size n for $\oplus\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is a fixed-capacity data structure that maintains values $a[1], \ldots, a[n] \in \mathbb{D}$ while allowing for updates to the values and queries for the aggregate value of any prefix and suffix. A FlatFAT instance is created by `new`$(\langle val_1, \ldots, val_n \rangle)$, which initializes $a[i] = val_i$ and sets the capacity to n. The reactive aggregator may invoke the following operations on a FlatFAT instance:

- `update`$(\langle (loc_1, val_1), \ldots, (loc_m, val_m) \rangle)$, where each loc_i is a unique location, writes val_i to $a[loc_i]$ for each i;
- `aggregate`() produces the result of $a[1] \oplus \cdots \oplus a[n]$;
- `prefix`(i) produces the result of $a[1] \oplus \cdots \oplus a[i]$; and
- `suffix`(j) produces the result of $a[j] \oplus \cdots \oplus a[n]$.

4.1 Design Overview

The reactive aggregator framework takes an incremental approach to maintaining the aggregate as the window changes. When only a small change is made to the window, the framework performs only a small amount of work. To realize this within the abstraction of lift, combine, and lower from Section 3, we maintain a number of partial aggregate results. When the window changes, the algorithm updates these results and uses them to derive the aggregate more efficiently than recomputing everything from scratch. In particular, we show the following bounds:

Theorem 4.1 *Let* lift, combine, *and* lower *be constant-time functions, and* Agg *be a constant-sized data type. The FlatFAT functions take $O(m + m\log(n/m))$ time, where m is the number of window events after the previous firing and n is the window size at the time of firing. Furthermore, FlatFAT consumes $O(n)$ space.*

Notice that this is $O(\log n)$ time for constant changes to the window and $O(m)$ for changes that completely overwrite the window, an amount that is already needed to make m changes. To meet these bounds, we first design FAT (fixed-sized aggregator), a data structure that acts as a container holding exactly n values while efficiently maintaining the aggregate of the contained data (Section 4.2). It is represented as a complete binary tree on n leaves, holding the window elements. We design algorithms to update and query this tree efficiently by maintaining, at each internal node, the aggregate of the data in the leaves below it.

An important feature of FAT is that it can be kept "flat" in consecutive memory in a layout so simple that it is pointerless (Section 4.3), motivated by the following observations about modern systems:

- *Pointers are expensive; eliminate them.* Pointers seem to be a necessity of complex data structures; however, they require significant memory just to keep and traversing pointers generally has performance implications. For example, each (system) pointer is 8 bytes on a 64-bit machine. For this reason, we employ (virtually) pointerless data structures.
- *Many small malloc's are slow; buy in bulk.* Dynamic memory allocation is typically faster when the allocation is performed in bulk than in multiple small requests of the same total size. Hence, we work with structurally-static data structures, where memory is only allocated once at creation or infrequently for resizing.
- *Cache misses are costly; place data carefully.* The memory hierarchy is complex to navigate, but it is a good principle to exploit spatial locality. Therefore, after we settled on using a tree, we picked a memory layout where sibling nodes that are accessed together are in the same cache line or consecutive lines.

4.2 Fixed-Sized Aggregator

We begin by describing the FAT tree, a conceptual data structure which provides a foundation for the description of the FlatFAT in the following section. The focus here is on how to efficiently support the functions `update`, `aggregate`, `prefix`, and `suffix` required by the reactive aggregator framework. Because this section deals with an abstraction implementation, we measure each function's cost in terms of the number of $\oplus$ operations, deferring the complexity analysis of internal bookkeeping to the next section. We assume that n is a power of two, which is how we will use it. The following theorem summarizes our results:

Theorem 4.2 *A size-n FAT can be maintained such that (i)* `new` *makes $n-1$ calls to $\oplus$, (ii) for m writes,* `update` *requires at most $m(1 + \lceil \log(n/m) \rceil)$ calls to $\oplus$, and (iii)* `prefix`(i) *and* `suffix`(j) *each requires at most $\log_2(n)$ calls to $\oplus$. Furthermore,* `aggregate`() *requires no $\oplus$ calls.*

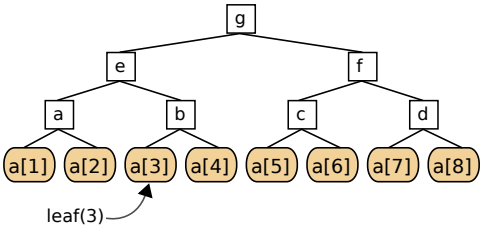

Figure 3: An example of FAT as a binary tree on 8 leaf nodes, storing $a[1], a[2], \ldots, a[8]$.

Algorithm 1: An algorithm for $\texttt{new}(\langle val_1, \ldots, val_n \rangle)$

```
1:  T ← Allocate a complete binary tree with n leaves
2:  for i = 1,...,n do T(leaf(i)) ← val_i
3:
4:  ℓ ← 1, W_1 ← {parent(leaf(i)) | i = 1,...,n}
5:  while (W_ℓ ≠ ∅) do
6:      for (v ∈ W_ℓ) do
7:          T(v) = T(left(v)) ⊕ T(right(v))
8:          if (v ≠ T.root) then
9:              W_{ℓ+1} = W_{ℓ+1} ∪ {parent(v)}
10:     ℓ ← ℓ + 1
11: return T
```

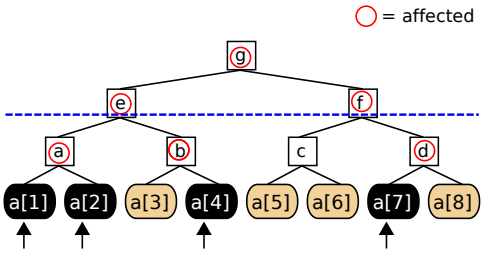

Figure 4: An 8-leaf FAT tree with $a[1], a[2], a[4]$, and $a[7]$ being updated. The internal nodes that need to be updated are circled. The dashed horizontal line marks the location of ℓ^* used in our analysis.

Algorithm 2: An algorithm for the operation $\texttt{update}(\langle (loc_1, val_1), \ldots, (loc_m, val_m) \rangle)$.

```
1: for i = 1,...,m do T(leaf(loc_i)) ← val_i
2:
3: ℓ ← 1, W_1 ← {parent(leaf(loc_i)) | i = 1,...,m}
4: while (W_ℓ ≠ ∅) do
5:     for (v ∈ W_ℓ) do
6:         T(v) = T(left(v)) ⊕ T(right(v))
7:         if (v ≠ T.root) then
8:             W_{ℓ+1} = W_{ℓ+1} ∪ {parent(v)}
9:     ℓ ← ℓ + 1
```

Representing FAT: We maintain FAT as a complete binary tree T with n leaves, which store the values $a[1], \ldots, a[n]$. The leaf node containing $a[i]$—or simply, the i-th leaf—is referred to by $\texttt{leaf}(i)$. Each internal node v keeps a value $T(v) \in \mathbb{D}$ that satisfies the invariant

$$T(v) = T(\texttt{left}(v)) \oplus T(\texttt{right}(v)), \tag{4.1}$$

where $\texttt{left}(v)$ and $\texttt{right}(v)$ denote the left child and the right child of v respectively. Thanks to associativity, simple mathematical induction implies that if v is the root of a subtree whose leaves are $a[j], a[j+1], \ldots, a[k]$, then

$$T(v) = a[j] \oplus \cdots \oplus a[k] \tag{4.2}$$

To illustrate Figure 3 shows a binary tree for FAT with 8 leaves. The leaf node corresponding to $a[i]$ is denoted by $\texttt{leaf}(i)$; for instance, as depicted in the figure, $\texttt{leaf}(3)$ refers to the leaf which stores $a[3]$. By definition, we know that $T(b) = a[3] \oplus a[4]$.

4.2.1 *Creating an Instance*

The user creates a new FAT instance by invoking `new` with the values val_i, $i = 1, \ldots, n$. Once invoked, `new` builds the tree structure and computes the value for each internal node, making sure that it satisfies equation (4.1).

Algorithm 1 shows the pseudocode for `new`. In words, it first allocates a complete binary tree containing n leaves and stores val_i in the i-th leaf (Lines 1–3). After that, it proceeds bottom-up level by level, computing $T(v) \leftarrow T(\texttt{left}(v)) \oplus T(\texttt{right}(v))$ for every internal node v. Because the computation for a level depends only on the computation of the levels below it, we know that equation (4.1) holds at all internal nodes after the algorithm finishes.

As an example, when `new` is called with $\langle x_1, x_2, \ldots, x_8 \rangle$, the algorithm first creates a tree structure like in Figure 3 and generates $W_1 = \{a, b, c, d\}$, where for each $v \in W_1$, it computes $T(v) \leftarrow T(\texttt{leaf}(v)) \oplus T(\texttt{right}(v))$. Then, it proceeds to work on $W_2 = \{e, f\}$ and $W_3 = \{g\}$. In this case, the number of $\oplus$ calls is $|W_1| + |W_2| + |W_3| = 4 + 2 + 1 = 7$.

In general, the cost of `new` in terms of the number of $\oplus$ calls can be analyzed as follows: For each level ℓ, the number of $\oplus$ calls is precisely $|W_\ell|$ as is evident from Line 7. Therefore, to obtain the total number of $\oplus$ calls, we only have to sum up the sizes of the W_ℓ's. Now observe that W_1—the sets of the parents of the leaves—is the set of all level-1 nodes and inductively, W_ℓ—the sets of the parents of $W_{\ell-1}$—is the set of all level-ℓ nodes, so $|W_\ell| = n/2^\ell$. Hence, we conclude that the number of $\oplus$ calls is $\sum_{\ell=1}^{\log_2 n} |W_\ell| = \sum_{\ell=1}^{\log_2 n} \frac{n}{2^\ell} = n - 1$, which follows from a standard geometric-sum identity. This proves clause (i) of Theorem 4.2.

4.2.2 *Updating Values*

The user modifies the contents stored in FAT by calling `update`. This takes a list of changes to be made and incorporates them into FAT by first updating the corresponding $a[\cdot]$ values and then updating the internal nodes affected by the changes. (Section 5.1.1 explains how to find these positions when a window slides.)

The goal is to update only the internal nodes that are affected by the changes. To understand this, consider the case where exactly one $a[i]$ is modified. Here only the internal nodes whose values depend on $a[i]$ need to be updated. The relationship in equation (4.2) characterizes which nodes need to be updated: exactly those on the path from $a[i]$ to the root. For example, by this reasoning, if $a[3]$ in Figure 3 is altered, only the nodes b, e, g need to be updated—and they should be updated in this order because of their dependencies.

By a similar reasoning, in the case of multiple modifications, an internal node needs to be updated if a leaf in its subtree is modified. There are, however, dependencies between these nodes that determine how we must update them. For instance, consider Figure 4, in which $a[1], a[2], a[4], a[7]$ are modified. Using this logic, we know that a, b, d, e, f, g are the nodes that have to be updated; these are circled in the figure. But although we may update a, b, and d in any order, we cannot, for example, update e before a and b.

To resolve the internal dependency, we follow the principle of change propagation [1] to identify and update the internal nodes affected by the modifications: when a node is updated, it triggers

the nodes that depend on it to be updated. Algorithm 2 summarizes the `update` algorithm. First, the algorithm modifies the leaves corresponding to the updates (Line 2). Then, it constructs W_1, the set of nodes in level-1 (one level above the leaves) that need to be updated. The rule is simple: if v is changed, $\texttt{parent}(v)$ will be updated. As shown in Lines 3–9, the algorithm proceeds bottom-up level by level, $\ell = 1, 2, \ldots$, updating the affected nodes W_ℓ, and scheduling the nodes that depend on them by adding them to $W_{\ell+1}$.

To better understand the `update` algorithm, we step through the algorithm using our running example in Figure 4. After updating the leaves, it identifies $W_1 = \{a, b, d\}$. Among these nodes, the algorithm does not prescribe which order they must be carried out, leaving the choice open to the implementation. While processing W_1, it populates W_2, resulting in $W_2 = \{e, f\}$. The same steps are repeated, yielding $W_3 = \{g\}$.

Correctness of the algorithm—that it ensures (4.1)—follows trivially by mathematical induction. The more involved question is, how many $\oplus$ calls are needed for an update of m locations?

We analyze the number of $\oplus$ calls by upper-bounding the number of calls per level of the tree. The number of calls at level ℓ, as is evident from the algorithm's description, is $|W_\ell|$. Hence, the total number of invocations is

$$\#\text{ of calls} = \sum_{\ell \geqslant 1} |W_\ell| \tag{4.3}$$

To proceed, we derive an upper bound on the size of each W_ℓ as a function of the number of modified leaves m and the level number ℓ:

Claim 4.3 *The work at level $\ell \in [1, \log_2 n]$ is $|W_\ell| \leqslant \min\{m, n/2^\ell\}$.*

Proof. Using the characterization in equation (4.2), we know that an internal node belongs to W_ℓ if and only if it is on the leaf-to-root path of a modified leaf. Because exactly m leaves were modified, there cannot be more than m leaf-to-root paths passing through level ℓ, so $|W_\ell| \leqslant m$. Furthermore, W_ℓ is, by definition, a subset of the nodes in level ℓ, so $|W_\ell| \leqslant n/2^\ell$. This means that $|W_\ell| \leqslant \min\{m, n/2^\ell\}$. □

Similar to [33], our analysis centers on accounting for work at different levels of the tree. Our analysis continues by separating the levels ℓ's into two parts—top and bottom. The top part accounts for level $\ell^* = 1 + \lceil \log_2(n/m) \rceil$ and above, and the bottom part accounts for levels below ℓ^*. Hence, # of calls = top + bottom, where

$$\mathsf{top} = \sum_{\ell=\ell^*}^{\log_2 n} |W_\ell| \quad \text{and} \quad \mathsf{bottom} = \sum_{\ell=1}^{\ell^*-1} |W_\ell|.$$

For intuition, these two cases are handled differently because most leaf-to-root paths have yet to merge near the leaves, whereas these paths have sufficiently merged in the top portion. The dashed horizontal line in Figure 4 illustrates this division.

We now analyze the contribution due to the top and the bottom, in turn: To analyze the top portion, let $\lambda = n/2^{\ell^*}$. Since $\lceil x \rceil \geqslant x$ for $x \geqslant 0$, it follows that $\lambda = \frac{n}{2^{1+\lceil \log_2(n/m) \rceil}} \leqslant \frac{n/2}{2^{\log_2(n/m)}} = \frac{n/2}{n/m} = m/2$. Therefore, we have

$$\begin{aligned}
\mathsf{top} = \sum_{\ell=\ell^*}^{\log_2 n} |W_\ell| &\leqslant \sum_{\ell=\ell^*}^{\log_2 n} \min\{m, n/2^\ell\} && \text{[Claim 4.3]} \\
&\leqslant \sum_{\ell=\ell^*}^{\log_2 n} \frac{n}{2^\ell} = \frac{n}{2^{\ell^*}} + \frac{n}{2^{\ell^*+1}} + \cdots + 1 && \text{[}\min\{a,b\} \leqslant b\text{]} \\
&\leqslant \frac{2n}{2^{\ell^*}} = 2\lambda && \text{[Geometric Sum]}
\end{aligned}$$

Algorithm 3: An algorithm for handling $\texttt{prefix}(i)$.

```
1: v ← leaf(i), a ← T(v)
2: while (v ≠ T.root) do
3:     p ← parent(v)
4:     if (v = right(p)) then a ← T(left(p)) ⊕ a
5:
6:     v ← p
7: return a
```

which means $\mathsf{top} \leqslant m$. The bottom part is simply

$$\mathsf{bottom} = \sum_{\ell=1}^{\ell^*-1} |W_\ell| \leqslant \sum_{\ell=1}^{\ell^*-1} m \leqslant (\ell^* - 1)m = \lceil \log_2(n/m) \rceil \cdot m.$$

Hence, we conclude that the total number of $\oplus$ calls is at most $m + \lceil \log_2(n/m) \rceil \cdot m$, proving clause (ii) of Theorem 4.2.

4.2.3 *Answering Queries*

The user can ask for the aggregate of the whole data, or any prefix or suffix using the `aggregate`, `prefix`, and `suffix` operations. The `aggregate` operation requires no work, as it only needs to return the value at the root of the tree. The `prefix` and `suffix` operations can be supported with minimal work because as we will show, any such query can be answered by combining at most $\log_2(n)$ values in the tree. We need the `prefix` and `suffix` operations later in the paper to correctly handle non-commutative aggregations.

We focus on the `prefix` operation; the `suffix` operation is symmetric. To gather some intuition, we consider answering `prefix`(7) on FAT in Figure 3. Instead of directly computing $a[1] \oplus \cdots \oplus a[7]$, we can break this up into a small number of segments that correspond to the nodes of T:

$$\begin{aligned}
a[1] \oplus \cdots \oplus a[7] &= (\underbrace{a[1] \oplus \cdots \oplus a[4]}_{T(e)}) \oplus ((\underbrace{a[5] \oplus a[6]}_{T(c)}) \oplus a[7]) \\
&= T(e) \oplus T(c) \oplus a[7].
\end{aligned}$$

Extending this idea, we present a natural bottom-up algorithm (Algorithm 3), which makes exactly one leaf-to-root traversal. It starts at the leaf node $\texttt{leaf}(i)$, setting $a = a[i]$ (Line 1). Then, as it walks up the tree, it incorporates more segments into a, extending the coverage of a further to the left. The main logic is on Line 4, which updates a to $T(\texttt{left}(p)) \oplus a$ if v is the right child of its parent. To understand this, it helps to notice that if v is the right child of $p = \texttt{parent}(v)$, node v's sibling—i.e., the left child of p—contains an extension of a, a new prefix segment next to what a has included.

In terms of complexity, the algorithm traverses a leaf-to-root path, making at most one $\oplus$ call at each node; therefore, the number of $\oplus$ calls is at most $\log_2(n)$, proving clause (iii) of Theorem 4.2. Furthermore, correctness of the algorithm follows from the invariant that after each execution of Line 4, a contains the aggregate of all the leaves to the left of $a[i]$ in the subtree rooted at p.

4.3 FlatFAT: Storing FAT in Memory

FlatFAT is an efficient implementation of the FAT data structure. Because FAT is structurally static, FlatFAT is able to allocate the necessary memory at creation and guarantee that sibling nodes are placed next to each other. This design reduces dynamic memory allocation calls in the overall framework and improves cache friendliness as these nodes tend to be accessed together.

To accomplish this, FlatFAT uses a numbering scheme often used in array-based binary heap implementations (see, e.g., [30]),

which we will now recall. Let T be a FAT of size n, so T is a tree with $2n-1$ nodes. This can be kept as an array of length $2n-1$ using the following recursive numbering[1]: the root node is at index $h(T.root) = 1$, and for a node v, its left and right children are at

$$h(\texttt{left}(v)) = 2h(v) \quad \text{and} \quad h(\texttt{right}(v)) = 2h(v)+1.$$

With this mapping, each of the navigation operations required by FAT takes $O(1)$ time. To navigate downward, we use the definition to give the positions of the left and right children of a node. To navigate upward, observe that for any node v other than the root, $h(\texttt{parent}(v)) = \lfloor h(v)/2 \rfloor$. Finally, to access a leaf node, we note that the location of the i-th leaf (i.e., the leaf corresponding to $a[i]$) is $h(\texttt{leaf}(i)) = n+i-1$.

Consequently, the following running time bounds follow directly from Theorem 4.2:

Corollary 4.4 *For a constant-time binary operator, a size-n FlatFAT can be maintained such that (i)* `new` *takes $O(n)$ time, (ii) for m writes,* `update` *takes $O(m + m\log(n/m))$ time, (iii)* `prefix`(i) *and* `suffix`(j) *each takes $O(\log_2(n))$ time, and (iv)* `aggregate`$()$ *takes $O(1)$ time.*

5. REACTIVE AGGREGATOR

As shown in Figure 2, RA accepts windowing events (insert, evict, trigger); internally uses the FlatFAT and aggregation functions; and finally implements the IStream operator from stream-relational algebra to submit output tuples. Upon an insert event, RA lifts the tuple using the lift function and stows it away in the FlatFAT. The lifted tuple remains there until the corresponding evict event. Meanwhile, RA can handle trigger events probing for the current window's aggregate value. It obtains these from the FlatFAT, lowers the result using the lower function, and calls submit on the SPL runtime to emit an output tuple.

So far, we have made no distinction between the instantaneous window size and the capacity of the FlatFAT, denoting both with n. For this section, we will be more explicit:

$$n_{\text{tuples}} = \text{the current number of tuples in the window; and}$$
$$n_{\text{capacity}} = \text{the current capacity of FlatFAT.}$$

Therefore, n_{tuples} changes as tuples enter or leave the window and n_{capacity} changes because of resizing.

5.1 Reactive Aggregator Using FlatFAT

To describe the RA implementation, we first describe how it maintains the window under tuple arrival and tuple eviction; then, we describe how it provides the aggregate upon request.

5.1.1 Maintaining a sliding window

RA views the slots of FlatFAT ($a[1], \ldots, a[n_{\text{capacity}}]$) as an array of length n_{capacity}, using this space to implement a circular buffer, where it keeps the (lifted) elements of the sliding window. As is standard, the implementation maintains a front pointer and a back pointer to mark the boundaries of the circular buffer. Unfilled FlatFAT slots are given a special marker, denoted by $\perp$. The $\perp$ marker serves as a neutral element that short-circuits the binary operator to return the other value: $x \oplus \perp = x$ and $\perp \oplus y = y$. This marker is not necessary for FIFO windows, as we will see in the following section.

At the start, RA creates a FlatFAT instance with a default capacity, filling all the slots with $\perp$. As tuples enter into the window, they are inserted into the circular buffer. As tuples leave the window, they are removed from the buffer, and their locations are marked with $\perp$.

[1]We use 1-based indexing.

As an example, Figure 5 shows an instance of a sliding window that keeps the latest 4 numbers, as well as how the window is physically represented. It also shows the locations of the front pointer (F) and the back pointer (B), which indicate, respectively, the starting point and the ending point of the circular buffer.

Unless the window logic respects FIFO semantics, the circular buffer can have "holes", potentially creating a situation where it cannot take more tuples even though there is room in the middle[2]. We resolve this with a `compact` operation. The `compact` operation starts by scanning the leaves, shifting non-$\perp$ values left to fill in holes marked by $\perp$ values, while building up a work list of parents whose children changed. After that, it uses the same algorithm as for `update` to rewrite internal nodes, see lines 3-9 of Algorithm 2. Overall, this requires time linear in the capacity of the buffer.

Despite compaction, windows may grow to fill up the FlatFAT or shrink such that usage is too low for the current capacity (see Section 2.2). We resolve this with a `resize` operation. The `resize` operation creates a new FlatFAT, and copies the data over, skipping $\perp$ values. In other words, it uses the algorithm for `new`, see Algorithm 1, which requires linear time in the capacity of the buffer.

Because of their hefty cost, the algorithm makes sure to call `resize` and `compact` judiciously so that their cost gets amortized over the normal window events. The algorithm works as follows:

- Upon receiving a tuple and the buffer is full: if $n_{\text{tuples}} \leqslant \frac{3}{4} n_{\text{capacity}}$, run `compact`; otherwise, use `resize` to double the capacity.
- After evicting a tuple: if $n_{\text{tuples}} < \frac{1}{4} n_{\text{capacity}}$, use `resize` to shrink the capacity by half.

Hence, we observe that (i) after a `resize` operation, the buffer is between $\frac{3}{8} n_{\text{capacity}}$ and $\frac{1}{2} n_{\text{capacity}}$ full; and (ii) after a `resize` or `compact` operation, the buffer has no holes. These observations are crucial in analyzing the amortized cost.

We begin by accounting for the cost of the `compact` operations. When a `compact` operation is performed, at least $\frac{1}{4} n_{\text{capacity}}$ evictions must have happened since the last time that there were no holes—because the buffer is full but $n_{\text{tuples}} \leqslant \frac{3}{4} n_{\text{capacity}}$. We charge the $O(n_{\text{capacity}})$ cost of compacting to the evictions that created these holes, $O(1)$ per eviction. We note that this accounting did not double charge because by (ii), there are no holes in the buffer after a `resize` or `compact` operation.

As for the `resize` operations, when the capacity needs to be doubled, at least $n_{\text{capacity}} - \frac{1}{2} n_{\text{capacity}} = \frac{1}{2} n_{\text{capacity}}$ arrivals must have happened since the last `resize`—because the buffer is full but immediately after the last `resize`, the buffer can only be between $\frac{3}{8} n_{\text{capacity}}$ and $\frac{1}{2} n_{\text{capacity}}$ full (via. (i) above). We charge the $O(n_{\text{capacity}})$ cost of doubling to these arrivals, $O(1)$ per arrival. By a similar reasoning, when the capacity is shrunk in half, at least $\frac{3}{8} n_{\text{capacity}} - \frac{1}{4} n_{\text{capacity}} = \frac{1}{8} n_{\text{capacity}}$ evictions must have happened since the last `resize`—because the buffer is $\frac{1}{4} n_{\text{capacity}}$ full but immediately after the last `resize`, the buffer can only be between $\frac{3}{8} n_{\text{capacity}}$ and $\frac{1}{2} n_{\text{capacity}}$ full (via. (i) above). Again, we charge the $O(n_{\text{capacity}})$ cost of shrinking to these evictions, $O(1)$ per eviction. Therefore, it takes constant amortized time to support an arrival or eviction event.

5.1.2 Reporting the aggregate result

We describe how RA derives the aggregate of the current window. At first glance, this seems to be just be a matter of reading a value: FlatFAT has the window contents stored in the leaves and its `aggregate`$()$ operation can return, at constant cost, the value of $a[1] \oplus \cdots \oplus a[n_{\text{capacity}}]$.

[2]We cannot put a new tuple in the holes because we have to retain the window order.

Event	Window's Contents	FlatFAT's Slots $a[1]$	$a[2]$	$a[3]$	$a[4]$
1) 4 arrives	4	F 4 B	$\perp$	$\perp$	$\perp$
2) 7 arrives	4, 7	F 4	7 B	$\perp$	$\perp$
3) 3 arrives	4, 7, 3	F 4	7	3 B	$\perp$
4) 2 arrives	4, 7, 3, 2	F 4	7	3	2 B
5) 4 leaves	7, 3, 2	$\perp$	F 7	3	2 B
6) 9 arrives	7, 3, 2, 9	9 B	F 7	3	2

Figure 5: A sequence of events that the window logic might issue in maintaining a sliding window that keeps the latest 4 numbers, showing how the window is physically represented.

However, more care is required to correctly derive the window aggregate. The problem, which we call the *inverted buffer* problem, is that the ordering in the linear space $a[1], a[2], \ldots, a[n_{\text{capacity}}]$ can differ from the ordering in the circular buffer (i.e., the window order). To explain this, we will revisit the example in Figure 5. Important to this discussion are the locations of the front pointer (F) and the back pointer (B), which indicate, respectively, the starting point and the ending point of the circular buffer. Events 1–5 are as expected: the window order is identical to FlatFAT's order. In event 6, the element 9 is inserted—correctly into $a[1]$; however, while the window order is $7, 3, 2, 9$, the "physical" order is inverted. Because of this, if we were to call `aggregate` now, we would get $9 \oplus 7 \oplus 3 \oplus 2$, which would be *incorrect* unless the operator is commutative.

The correct aggregate in this case is not difficult to derive. The key is to recognize that an inverted buffer is a circular buffer split in the middle because of the linear address space. Hence, the correct aggregate is `suffix`(F) $\oplus$ `prefix`(B), which FlatFAT can answer.

Therefore, to report the aggregate result, we first detect whether the buffer is currently inverted. The buffer is inverted only if the back pointer has wrapped around and comes up ahead of the front pointer. In other words, it is inverted if and only if the back pointer precedes the front pointer. If the buffer is *not* inverted, we use `aggregate`(); if the buffer *is* inverted, we use `suffix`(F) $\oplus$ `prefix`(B). In either case, this costs at most $O(\log_2 n_{\text{capacity}})$ time.

5.2 Optimizing For FIFO

Many sliding-window policies are first-in, first-out (FIFO): the first tuple to arrive is the first to leave the window. Indeed, in SPL, both count-based and time-based policies ensure FIFO ordering. We describe some optimization for this common case.

When the window is FIFO, the area from the front pointer to the back pointer (wrapping around the array boundary) is always fully occupied—there can be no "holes." The direct consequence for RA is that it does not need a `compact` operation, making it easier to maintain the desired utilization. But more importantly, this means that we do not need to explicitly store $\perp$ in unused slots: the buffer's demarcation can be baked into the `update` algorithm, so unused slots are automatically skipped over. When the buffer is inverted, the unoccupied area is between the leaf-to-root path of the back pointer and that of the front pointer. Symmetrically, when the buffer is normal, the occupied area is between the leaf-to-root path of the front pointer and that of the back pointer.

5.3 Optimizing Through Code Generation

We use operator code generation to reduce runtime overhead by moving many decisions to compile time. This has an important benefit of static type checking. Through SPL's code-generation interface [18], operators can generate, at compile time, code specialized for a particular configuration. Because in our scenario, data types (schemas) and aggregation functions are known in advance, we are able to use this interface to hardwire both types and functions while ensuring type safety. This helps eliminate overhead from dynamic dispatching and runtime handling of variable types, and facilitates inlining. We remark that our baseline operators (both SPL's existing implementation [18] and our implementation of Arasu and Widom's algorithm [3]) also use this approach.

5.4 Additional Features

Users often want to aggregate per group based on one or more key attributes rather than globally over an entire window. We support this by maintaining a separate instance of FlatFAT for each active unique key, specifically using the partitioning feature of SPL's windowing library [12]. Furthermore, because RA is a bona fide operator in a general-purpose streaming system, it can be combined with other technologies on that streaming system. For instance, other work shows how to auto-parallelize SPL operators [29], and those techniques apply directly to RA with group-by. As another example, while Aggregate itself is a unary operator, SPL can assemble operators into arbitrary topologies supporting, among other things, multiple input streams [18].

6. EVALUATION

This section presents an experimental evaluation that examines whether the theoretical complexity bounds for RA are true in practice, as well as whether the raw performance of the RA framework is competitive. Our baseline is the standard library implementation of Aggregate in IBM InfoSphere Streams (abbreviated *ISS*), which is nonincremental. We refer to reactive aggregation as *RA*. We also study RA's performance relative to an implementation of Arasu and Widom's algorithm (henceforth, *AW*) [3], the only prior algorithm that matches RA's theoretical complexity.

6.1 Experimental Setup

The performance of sliding-window aggregation depends on the window size n; we intuitively expect it to spend more time on larger windows. All figures in this section put the window size on the x-axis. To evaluate a large range of sizes, we use a logarithmic scale. Unless otherwise specified, all experiments use a slide granularity $m = 1$, i.e., each time the aggregation fires, it handles one insertion and one eviction. As will be explained later, this is the worst-case behavior in terms of per-tuple cost. Furthermore, we use 32-bit integers as the data type to aggregate unless specified otherwise.

All experiments used IBM InfoSphere Streams and the SPL language. While SPL programs are usually distributed across a cluster of machines, for our experiments, we wanted to minimize interference from distribution unrelated to the research questions at hand. Hence, we built each benchmark as a standalone process that runs on a single machine. For robustness, we performed *three* independent runs, each flowing roughly 120 million tuples through; we report the median time and the maximum memory consumption.

We ran experiments on a 2-core, 3.00 GHz Intel Xeon X5160 with 8 GB of RAM. The L2 data cache is 4 MB. The machine ran RHEL 6.3, and we used GNU g++ 4.4.6.

6.2 Results and Analysis

Break-Even Points: The first question we investigate is, *what is the smallest window size at which incremental computation in our RA framework pays off?* With very small windows, ISS does so little work that it outperforms an incremental scheme; with large windows, ISS eventually becomes so expensive that it will be outperformed by incremental evaluation. We ran both implementations at different window sizes and measured the sustained throughput until we narrowed the break-even point down to the nearest 10.

Aggregation Function	Break-Even Point	10x
Count	Never	Never
Sum	≈ 370	≈ 5,200
Sum<Double>	≈ 290	≈ 5,200
Max	≈ 260	≈ 5,200
Max<Double>	≈ 130	≈ 3,600
ArithmeticMean	≈ 10	≈ 900
MinCount	≈ 200	≈ 4,480
SampleStdDev	≈ 10	≈ 700
PopulationStdDev	≈ 10	≈ 700
ArgMax	≈ 130	≈ 2,770
ArgMax<Double>	≈ 250	≈ 5,810
Collect	Never	Never

Table 3: The smallest window size (to the nearest ten), for different aggregation functions, where incremental computation in RA becomes faster than ISS and RA becomes 10x faster than ISS, which reevaluates the whole window.

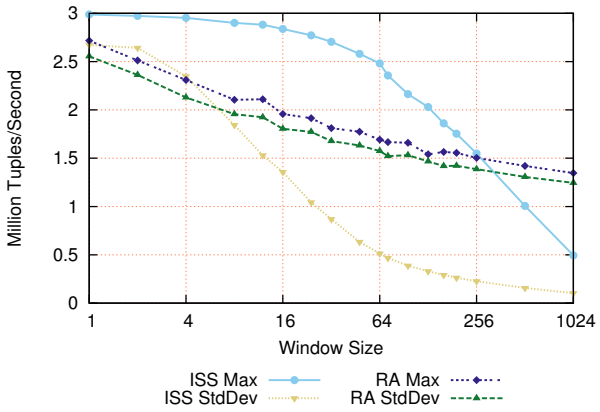

Figure 6: The sustained throughput (in **million tuples per second**) for different sliding-window aggregate schemes (ISS vs. RA) and aggregation functions (Max and StdDev) as the window size is varied.

Table 3 shows the results. A lower number represents a better result for our new algorithm. All experiments use 32-bit integers except for those annotated with <Double>, which use 64-bit floating point numbers. We observe break-even points from 10 to 370 depending on the inherent cost of the aggregation functions. RA does not break even for Count or Collect. The baseline Count simply reads off the container size, which is a constant-time operation. Collect computes an output of the same size as the window: although both ISS and RA perform the same $n-1$ operations, the framework overhead in RA is, as to be expected, higher than that of a reexecution scheme. In all other cases, RA breaks even for relatively small windows—and in general, the costlier the aggregation function, the sooner RA breaks even.

Column 10x in Table 3 shows the smallest window size at which RA outperforms ISS by one order of magnitude. This happens at relatively moderate window sizes between 700 and 5,810, confirming the practical utility of our framework on reasonable window sizes.

Relative Performance: *How much slower is RA on small windows, and how much faster is it on large windows than reexecution?* Figure 6 shows the sustained throughput of the two schemes at different window sizes (higher is better). There are two pairs of lines. The two Max lines show that RA is about 10% slower on small windows and almost 3x faster on windows of size 1,024. The two StdDev lines show that RA is about 5% slower on small windows and over 10x faster on windows of size 1,024. These numbers demonstrate that the overhead is moderate on small windows; we are not concerned with the slight slowdown, because at such small

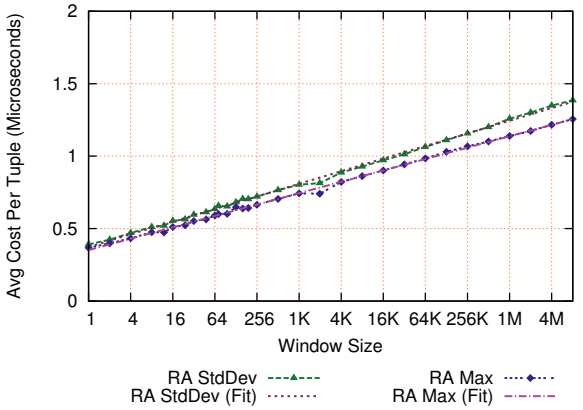

Figure 7: The average per-tuple cost (in **microseconds**) to update after a $(1,1)$-change for the Max and StdDev aggregation functions. Both the measured values and their best-fit to the log curve are shown.

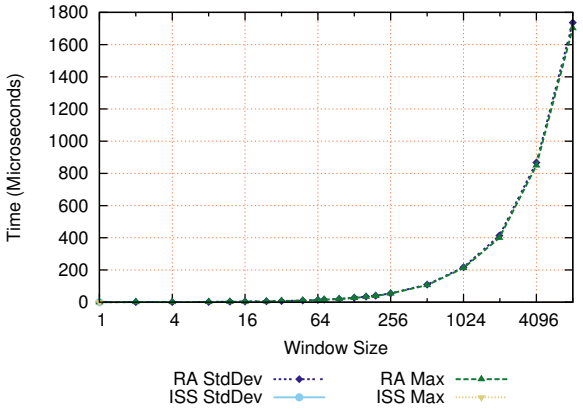

Figure 8: The time (in **microseconds**) to aggregate the whole window from scratch without incremental computation (i.e., the standard non-RA implementation) and with incremental computation for the Max and StdDev aggregation functions.

window sizes, the aggregation is unlikely to be the bottleneck of the application anyway. On the other hand, RA yields large speedups even at the moderate window size of 1,024, and we can observe even larger speedups when increasing the window size further.

Time Complexity: *Are the theoretical time complexity bounds true in practice?* Figure 7 shows the average per-tuple cost of RA at different window sizes, and compares it to the ideal line. Note that since the x-axis uses a log-scale, a straight line actually indicates logarithmic performance. Given that the measured results closely match the ideal line, we can conclude that the performance is indeed $O(\log n)$ as predicted. Furthermore, note that the x-axis goes up to a window size of 8 million tuples. This requires RA to store 16 million intermediate results. Each intermediate result for Max takes 4B, and each intermediate result for StdDev takes 20B. This means the memory footprint of the entire data structure is 64MB and 320MB, respectively, far exceeding the size of all caches and TLBs on the machine where we ran our experiments. However, despite that, there is no knee in the curve at the critical memory-hierarchy boundaries, because our algorithm only touches a small part of the data structure each time it fires.

Tumbling Windows: *How does RA perform when there is no opportunity for it to pay off because the window does not slide incrementally?* This is a measure of the framework's overhead. Let n be the window size and m be the sliding granularity, i.e., the number

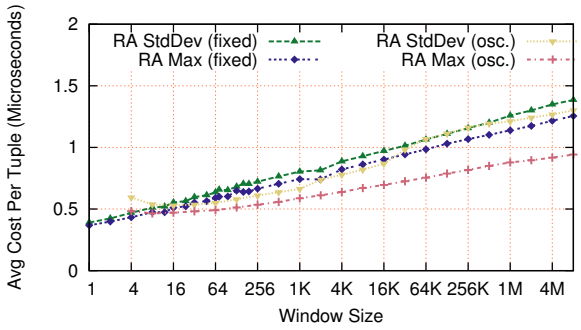

Figure 9: The average tuple cost (in **microseconds**) for the reactive aggregator to compute Max and StdDev with fixed window capacities of size n (*fixed*), and window capacities that oscillate between n and $2n$ (*osc.*).

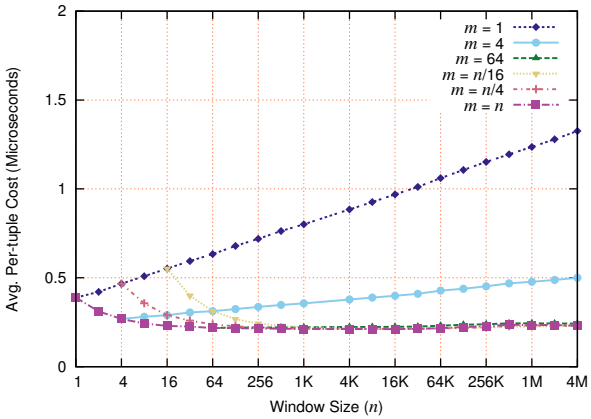

Figure 10: The average per-tuple cost (in **microseconds**) to update a batch of m changes using the reactive aggregator (RA) with StdDev, where the window size is n.

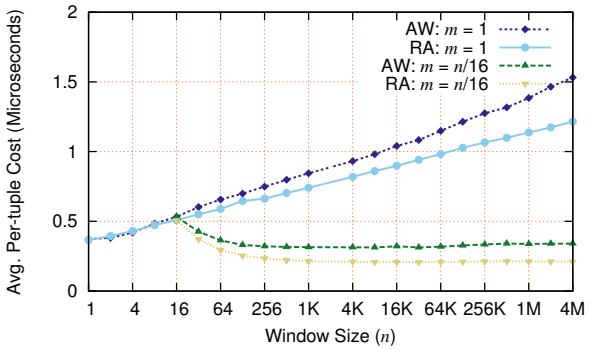

Figure 11: The average per-tuple cost (in **microseconds**) for Max to incorporate m changes, using the reactive aggregator (RA) vs. using the algorithm by Arasu and Widom (AW), as the window size (n) is varied.

of tuples inserted and removed on each aggregation firing. The experiments so far used $m = 1$, resulting in fine-grained sliding. On the other hand, when $m = n$, the sliding window degenerates to a tumbling window. None of the results can be reused from one firing to the next. Figure 8 shows the average cost per firing in microseconds (lower is better). Evidently, the cost is about the same irrespective of the scheme (RA or ISS) or the aggregation function. This indicates that RA has a low overhead: even when RA cannot benefit from incremental changes, it performs essentially no worse than the non-incremental algorithm.

Dynamic Windows: *How does RA perform when it must constantly resize its window's capacity?* Recall from Section 5.1.1 that if the buffer is full when a tuple arrives ($n_{\text{tuples}} = n_{\text{capacity}}$), we call `resize` to make $n'_{\text{capacity}} = 2n_{\text{capacity}}$. Upon evicting a tuple from the window, if $n_{\text{tuples}} < \frac{1}{4}n_{\text{capacity}}$, we call `resize` to make $n'_{\text{capacity}} = \frac{1}{2}n_{\text{capacity}}$. Figure 9 shows experiments designed to force the window capacity to oscillate between n and $2n$. First we send enough tuples to ramp up the capacity so that $n_{\text{capacity}} = 2n$. We then evict $\frac{1}{2}n$ tuples, forcing a call to `resize`, causing $n'_{\text{capacity}} = n$. The process then starts over after the window receives the next $\frac{1}{2}n$ tuples. The data confirms that the cost of resizing the capacity is indeed amortized. The oscillating experiments are faster because they spend most of their aggregation time with $n_{\text{tuples}} < n$, while the non-oscillating experiments spend all of their aggregation time with $n_{\text{tuples}} = n$. This discrepancy, however, is necessary if we are to force the capacity to vary, as the resizing scheme is explicitly designed to amortize the linear cost of a resize.

As with any solution whose algorithmic complexity depends on amortization, individual firings may incur a linear cost, increasing latency. For example, for StdDev, resizing from a window of size 256 to 512 takes 4.2 μs, and resizing from 1 M to 2 M takes 9.4 ms. Circumstances that cannot tolerate a rare linear cost can fix the window size to a large value of n to ensure no resizing.

Coarse-Grained Sliding: Often, multiple window events take place between firings. *How does RA behave between the two extremes of fine-grained sliding and tumbling?* Figure 10 shows the average per-tuple cost (lower is better), with separate curves for different sliding granularity m. Recall that the theoretical time complexity of our algorithm is $O(m + m\log(n/m))$ per firing. Given that each firing processes m tuples, that amounts to $O(1 + \log(n/m))$ per tuple. For constant m, this is the same as $O(\log(n))$, and indeed, the lines for $m = 1$, $m = 4$, and $m = 64$ are straight on a log-scale. On the other hand, when m is a fraction of n, the time complexity becomes $O(1 + \log(n/n))$, in other words, a constant. The lines for $m = n/16$, $m = n/4$, and $m = n$ indeed demonstrate constant per-tuple cost, for large enough window sizes, as predicted. The downward trend on small windows can be attributed to the overhead due to change propagation's walking up the tree, about $O(\log n)$ per firing, which gets absorbed in the big-O notation in the complexity bound and is offset by the cost of real aggregation operations for larger window sizes.

Comparison with Arasu-Widom: *How does RA compare to prior work?* In prior work, Arasu and Widom (AW) [3] describe the only algorithm that matches RA's theoretical complexity bounds. We re-implemented AW in InfoSphere Streams and added the same code-generation optimization we used for RA. As Figure 11 shows, both implementations have comparable algorithmic complexity in practice, with RA faster by a constant factor. We further confirm this across a range of aggregation functions, as Table 4 shows: for the same window size, the choice of an aggregation function makes little difference to the relative performance. However, inherent differences in the data representation seem to contribute to the widening of the gap as the window size increases. Whereas RA uses a single flat array, AW uses several arrays. Also, whereas RA uses simple loops to derive the aggregated value, AW contains a relatively-more-complicated step that splits the current window into base intervals.

Space Complexity: *What difference does RA make for space consumption?* Figure 12 shows the resident set size (RSS; lower is better), which we measured using a separate process that polls the `/proc` file system once a second. Note that this methodology did not interfere with the overall timing results, from which we conclude that it did not perturb the experiment. The figure only shows results for windows up to 4K entries, because ISS is very slow beyond

Aggregation Function	Relative Time T_{AW}/T_{RA}		
	$n = 8$	$n = 8K$	$n = 8M$
Sum	1.02	1.20	1.36
Max	1.02	1.25	1.37
ArithmeticMean	1.05	1.24	1.39
MinCount	1.04	1.23	1.35
PopulationStdDev	1.06	1.26	1.37

Table 4: Relative average per-tuple cost T_{AW}/T_{RA} (ratio) for firing after every tuple's arrival, where T_{AW} is the time for Arasu-Widom and T_{RA} is the time for our reactive aggregator.

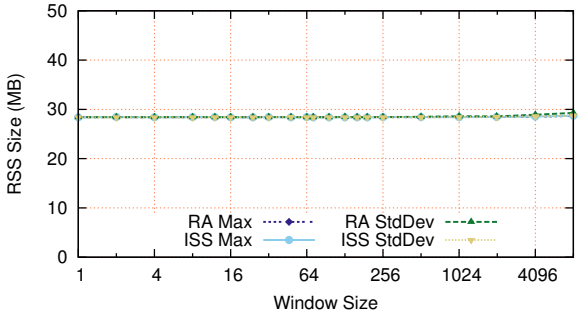

Figure 12: The resident set size (RSS, in **MB**) of the program that maintains Aggregate comparing ISS and RA.

4K entries. It confirms our theoretical result that RA requires $O(n)$ space and that the constant is small. At these small to moderate window sizes, the impact of window size on RSS is minor compared to the base RSS of the entire process, including the SPL runtime. Beyond 4K entries, the RA version continues to consume space linear in the size of the window.

Conclusion: Our experiments confirm the theoretical algorithmic complexity and space bounds. Furthermore, our experiments show that the practical performance of RA is competitive for small windows and much better than reexecution for large windows.

7. RELATED WORK

This section compares and contrasts our Reactive Aggregator (RA) with related work on sliding-window aggregation in particular and incremental computation in general.

7.1 Sliding-Window Aggregation

Gray et al. propose a trichotomy of aggregation functions [14], and many sliding-window aggregation papers adopt this terminology. An *algebraic* aggregation is associative and has fixed-sized intermediate results. A *distributive* aggregation is a special case of algebraic aggregation that does not require a final lower step. And a *holistic* aggregation has no constant bound on the size of intermediate results. This trichotomy was initially intended to explain implementation strategies for data cubes. We did not adopt it, because it does not capture invertibility or commutativity, which are important for sliding windows on ordered streams.

In the following discussion, let n be the window size, m the number of updates in a batch, and s the size of the stream history. In robust, long-running systems, s tends to infinity.

Work on temporal databases predates stream processing but addresses issues similar to sliding-window aggregation. Temporal databases store $O(s)$ tuples and support aggregation over any historical windows, whereas streaming systems store only $O(n)$ tuples and only support aggregation over windows that end at or near the current time. Moon et al. implement aggregation using red-black trees [26], and Yang and Widom use SB-trees, a hybrid of segment trees and B-trees [36]. For invertible aggregations, these temporal aggregation algorithms emulate evictions as negative insertions, yielding time $O(\log n)$ for $m = 1$. However, for non-invertible aggregations, they construct trees over the complete stream history, yielding time $O(\log s)$. In contrast, RA yields time $O(\log n)$ for $m = 1$ even for non-invertible aggregations.

Bulut and Singh save time and space with an approximate aggregation algorithm based on wavelets [8]. RA, in contrast, is an exact algorithm. Gigascope splits aggregates to support pre-aggregation on network interface cards [9] but does not use sliding windows. Hirzel aggregates incrementally over patterns instead of windows [17]. RA handles sliding windows, which are tricky because of tuple eviction. Ghanem et al. incrementalize a broad spectrum of sliding-window queries using negative tuples [13]. Unlike RA, this means they require invertible functions.

Arasu and Widom present the B-Int algorithm for incremental aggregation using base intervals [3]. B-Int is the only algorithm we are aware of that matches the algorithmic complexity of RA and handles non-invertible aggregations. However, we made RA more general as it is intended as a drop-in replacement for the aggregation operator in a commercial streaming system. In particular, B-Int does not address non-FIFO windows or variable-sized windows.

Some algorithms optimize for large m, by using only a two-level aggregation instead of a full-fledged tree. Examples include paned windows [23] and paired windows [22]. These approaches use time $O(n/m)$ for each update, degenerating to $O(n)$ for small m. RA's algorithmic complexity matches that of paned and paired windows for large m, but it is better for small m.

To our knowledge, none of the existing algorithms can both match RA's time complexity and support variable-sized windows.

The idea of decomposing aggregation operations into multiple helper functions is not new. For instance, it is a key feature of the ATLAS extension to SQL [35]. Yu et al. even offer techniques for automatically decomposing aggregation operations [37]. Some systems decompose aggregation operations differently than RA. Specifically, RA uses a combine(Agg,Agg) function, resembling a monoid [16]. Other systems (including ATLAS [35]) use an addOne(In,Agg) function, resembling a fold in functional programming. The drawback of a fold is that it adds items one by one in linear time; in contrast, the monoid variant enables RA to break down a window into balanced subwindows that it aggregates in logarithmic time.

This work focuses on *incremental* aggregation. An orthogonal topic is *shared* aggregation, optimizing the case where many similar stream queries are evaluated together. While some of the papers discussed above offer both incremental and shared aggregation [3, 22], we leave shared aggregation with RA to future work since our customers have yet to request it.

Soisalon-Soininen and Widmayer [33] study a class of search trees known as stratified trees and propose an algorithm for bulk insertion, for which they study rebalancing cost in terms of pointer changes. In another paper, Soisalon-Soininen and Widmayer [32] study the complexity of bulk insertion into an AVL tree. These papers, like ours, consider bulk operations and show the benefits of coordinating bulk changes to start from the leaves and progress up to the root. However, both their goals and techniques differ significantly from ours. Their analyses are tailored for pointer-update cost due to rebalancing and are not readily applicable to our setting, which keeps no pointers and requires no rebalancing but where the main cost is in maintaining the partial aggregation values.

7.2 General Incremental Computation

Whereas RA incrementalizes sliding-window aggregation, there is also literature on incrementalizing more general computations.

Demers et al. incrementalize attribute grammars using static dependency graphs [10]. The RETE algorithm incrementalizes production rule matching, and can handle not just changes in data, but also changes in rules themselves [11]. Later, Pugh and Teitelbaum use result memoizing to incrementalize pure functions [28]. None of these approaches focuses on aggregation.

Functional reactive programming (FRP, [19]) and self-adjusting computation (SAC, [1]) incrementalize general programs in functional languages. Both build and use dynamic dependency graphs, yielding a high overhead in both time and memory consumption. In the ideal case, these dynamic dependency graphs turn out to be balanced trees, yielding logarithmic performance. In contrast, the trees that RA constructs are leaner and always balanced.

There is growing literature on incremental large-scale batch processing. In contrast to our work on streaming with latencies in the micro-second range, these systems focus on scaling out to large clusters and have several orders of magnitude slower latencies. CBP (continuous bulk processing) makes it easier to hand-incrementalize computations by externalizing state as loop-back I/O in functional computation stages [24]. The Percolator uses observers, which are similar to database triggers, over a distributed data store [27]. Nectar offers a caching service where derived datasets are indexed by the computation that produced them [15]. Incoop [5] and Slider [6] memoize partial results of Hadoop jobs and use the associativity of Hadoop's combine functions for tree-based aggregation. Similar to our work, Slider uses a circular data structure for fixed-size sliding windows. But unlike our work, it does not keep trees balanced for variable-size windows. M3R (main-memory map reduce) focuses on caching results in iterative jobs, rather than incremental jobs where the input changes [31]. Finally, differential dataflow in Naiad uses caching for both iterative and incremental jobs [25] by using multi-dimensional version numbers. All of these discretize the computation into large batches, whereas RA can efficiently handle even single-tuple updates.

8. CONCLUSION

We have described Reactive Aggregation (RA), a new framework for sliding-window aggregation that works for non-commutative or non-invertible aggregations and even on non-FIFO windows. For m changes in a size-n window with $O(1)$-time aggregation functions, it updates results in $O(m+m\log(\frac{n}{m}))$ time. By using a flat, pointerless structure and code generation, RA achieves outstanding raw performance.

References

[1] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(1), 2009.

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

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

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *Symposium on Cloud Computing (SoCC)*, 2011.

[6] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental sliding-window computations for large-scale data analysis. Technical Report MPI-SWS-2012-004, Max Planck Institute for Software Systems, 2012.

[7] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. In *Very Large Data Bases (VLDB)*, pages 232–243, 2010.

[8] A. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization in large networks. In *International Conference on Data Engineering (ICDE)*, pages 303–314, 2003.

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

[10] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages (POPL)*, pages 105–116, 1981.

[11] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[12] B. Gedik. Generic windowing support for extensible stream processing systems. *Software Practice and Experience (SP&E)*, 44(9):1105–1128, 2014.

[13] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):57–72, 2007.

[14] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *International Conference on Data Engineering (ICDE)*, pages 152–159, 1996.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[16] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming (JFP)*, 16(02):197–217, 2006.

[17] M. Hirzel. Partition and compose: Parallel complex event processing. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 191–200, 2012.

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

[19] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University*, 2003.

[20] IBMInfoSphereStreams. IBM InfoSphere Streams. http://www.ibm.com/software/data/infosphere/streams/. Accessed: 2014-08-27.

[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbets, and S. Zdonik. Towards a streaming SQL standard. In *Very Large Data Bases (VLDB)*, pages 1379–1390, 2008.

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

[23] 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.

[24] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Symposium on Cloud Computing (SoCC)*, pages 51–62, 2010.

[25] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

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

[27] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2010.

[28] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[29] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[30] R. Sedgewick. *Algorithms in C++ - Parts 1–4: Fundamentals, Data Structures, Sorting, Searching (3. ed.)*. Addison-Wesley-Longman, 1999.

[31] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. In *Very Large Data Bases (VLDB) Industrial Track*, pages 1736–1747, 2012.

[32] E. Soisalon-Soininen and P. Widmayer. Amortized complexity of bulk updates in avl-trees. In *SWAT 2002, 8th Scandinavian Workshop on Algorithm Theory*, pages 439–448, 2002.

[33] E. Soisalon-Soininen and P. Widmayer. Single and bulk updates in stratified trees: An amortized and worst-case analysis. In *Computer Science in Perspective, Essays Dedicated to Thomas Ottmann*, pages 278–292, 2003.

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

[35] H. Wang, C. Zaniolo, and C. R. Luo. ATLAS: A small but complete SQL extension for data mining and data streams. In *Demonstration at Very Large Data Bases (VLDB-Demo)*, pages 1113–1116, 2003.

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

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

[38] 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.