Short paper

SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries

Georgios Theodorakis
Imperial College London
grt17@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

Holger Pirk
Imperial College London
pirk@imperial.ac.uk

ABSTRACT

Aggregate window computations lie at the core of online analytics in both academic and industrial applications. To efficiently compute sliding windows, the state-of-the-art algorithms utilize incremental processing that avoids the recomputation of window results from scratch. In this paper, we propose a novel algorithm, called SlideSide, that extends `TwoStacks` for multiple concurrent aggregate queries over the same data stream. Our approach uses different yet similar processing schemes for invertible and non-invertible functions and exhibits up to 2× better throughput compared to the state-of-the-art incremental techniques in a multi-query environment.

1 INTRODUCTION

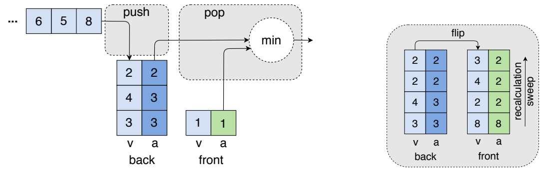
An ever-growing amount of data needs to be analyzed in real-time. Applications ranging from credit card fraud detection to clickstream analytics are not supported by "classic" relational systems and algorithms. Consequently, streaming applications have become increasingly important. One of the key operators in stream processing is window aggregation [1], i.e., the calculation of running aggregates over the continuous data stream.

Since data streams are conceptually infinite, they are partitioned into finite subsets of elements, called *windows*. A window has a *definition*, which maps each input tuple to a window *instance*. Upon aggregation, each window instance yields a result. Windows can be distinguished by whether their instances are disjoint ("tumbling windows") or not ("sliding windows"). Tumbling (a.k.a. fixed) windows slice up the input stream into segments with a fixed size temporal length (static *window size*). Sliding (a.k.a. hopping) windows generalize tumbling windows by specifying a *slide* parameter in addition to the size that specifies the distance between the start of two windows.

While tumbling windows are amenable to classic "relational" queries implementation techniques, the performance of sliding windows is more challenging to compute efficiently. Incremental algorithms introduce inherent control dependencies in the CPU instruction stream, as intermediate results from previous window instances have to be used to compute efficiently the next result. This is amplified in the case of multiple-queries applying computations over the same data stream, which has not been explored comprehensively. An example of the latter scenario is a live-visualization dashboard that plots line charts of aggregates on time-series data at different zoom levels [9].

Our contributions are the following:

- We study the performance of the best-performing incremental algorithms, as reported in recent literature [4, 6]. We determine sections of the problem space in which different approaches perform best (focusing specifically on multi-query processing).



(a) Pushing and Popping (b) Flipping

Figure 1: The TwoStacks algorithm

- We propose a novel approach for incremental processing in multi-queries scenarios, called SlideSide. Our solution extends the logic of `TwoStacks` [4] based on the insight that the algorithm maintains a running `preffix-/suffix-scan` over the input stream. SlideSide optimizes its performance for associative aggregation functions and can serve as a drop-in replacement for the aggregation operator in a streaming system.
- Finally, we demonstrate that SlideSide is competitive with highly optimized single-query algorithms, while it yields up to 2× better throughput and comparable latency in the multi-query scenario.

The remainder of the paper is organized as follows: in Section 2 we survey the state-of-the-art in incremental window computation. Section 3 introduces our novel incremental algorithm. The paper finishes with evaluation results (Section 4), and conclusions (Section 5).

2 BACKGROUND

In this section, we provide background on the underlying concepts of incremental processing. We also review current approaches and provide in Table 1 a summary of their complexities.

Algebraic Properties. In this work, we focus on associative algebraic aggregations [3] and consider the following properties:

- Invertibility: $(x \oplus y) \ominus y = x, \forall\, x, y$. This property can be exploited by introducing the following function: `inverse(a: Agg, b: Agg): Agg`, which removes the oldest partial aggregate from the window result with an incremental operation.
- Commutativity: $x \oplus y = y \oplus x, \forall\, x, y$

Partial Aggregates are smaller units of computation that compose the aggregate functions. Partial aggregation allow us to buffer and apply inexpensive aggregation and trivially parallelize the computation (e.g., using SIMD instructions [8]), when there are no data dependencies. This idea is applied in the form of *window slicing* [9] over a stream of data, where a slice is defined as the largest sequence of tuples that offer no sharing potential.

Incremental Aggregation Techniques. After the computation of partial results, the final aggregation step has to be applied to generate the query results. For that, streaming systems utilize the incremental aggregation techniques we describe next and summarize in Table 1.

Subtract-on-Evict (SoE) [4] is the best-performing approach in the case of invertible functions, i.e., AGG_{sum}. With SoE, the result of the previous window instance is re-used to compute the next

Algorithm		Time				Space	
		Single Query		Multi-Queries		Single-Query	Multi-Queries
		Amort	Worst	Amort	Worst		
SoE [4]	Inv	2	2	q	q	n	qn
	Non-Inv	n	n	qn	qn	n	qn
FlatFAT [7]		$log(n)$	$log(n)$	$qlog(n)$	$qlog(n)$	$2n$	$2n$
TwoStacks [4]		3	n	q	qn	$2n$	$2qn$
Slick Deque [6]	Inv	2	2	$2q$	$2q$	n	$q+n$
	Non-Inv	<2	n	q	qn	2 to $2n$	2 to $2n$
SlideSide	Inv	3	n	q	q	3n	$3n$
	Non-Inv	3	n	q	qn	$2n$	$2n$

Table 1: Algorithmic Complexities (partial aggregates: n, queries: q)

in constant time, by removing the expired elements and merging in the new data. However, SoE cannot efficiently compute non-invertible functions (e.g., AGG_{min}), as the whole window needs to be rescanned in the worst-case scenario.

TwoStacks [4] can be used for non-invertible functions. Figure 1 illustrates an example of the TwoStacks algorithm, which maintains a *back* and a *front* stack to store the input values and the aggregates required to produce the window results. When a new input value v arrives, its aggregate is computed based on the value of the back stack's top element and it is **push**ed onto the back stack. For every **pop** operation, the top from the front stack is removed and a result is produced by aggregating its value with the top of the back stack. Whenever the front stack is empty, the algorithm **flip**s the back onto the front stack, reversing the order of the values and recalculating the aggregates. However, as this happens infrequently, it exhibits $O(1)$ amortized complexity.

Multi-Query Algorithms. The previous algorithms are not designed to efficiently share intermediate results between multiple window definitions over the same stream, in contrary to approaches such as FlatFAT [7] and SlickDeque [6]. FlatFAT uses a pointer-less binary tree structure to store the partials, which results in $O(logn)$ complexity for a single query. SlickDeque proposes a different solution for invertible and non-invertible functions. For invertible functions, SlickDeque generalises SoE to answer multiple queries, by maintaining multiple instances of the original algorithm with partials that share the same memory space. In the case of non-invertible functions, instead of using a queue implemented by two stacks, SlickDeque uses a deque structure for insertions/removals of aggregates and answering queries with $O(1)$ amortized complexity. The time and space complexities of the non-invertible functions (see Table 1) depend on the input, with the worst-case scenario being a stream that is ordered in the opposite way of the aggregate operator order (i.e., if AGG_{max} the input is ordered descendingly).

3 SLIDESIDE

Let us now, describe SlideSide, our novel algorithm for accelerating incremental aggregation in a multi-query environment. It aggressively reuses intermediate results with data structures that have a sequential memory layout. Fundamentally, SlideSide is an extension of the TwoStacks algorithm. However, it uses different processing schemes for invertible and non-invertible functions.

Regarding the algebraic properties of the aggregate functions, SlideSide has the same requirements as the state-of-the-art algorithms described in Section 2 (associative aggregate functions). SlideSide can be applied to FIFO windows (in-order data).

3.1 Invertible Aggregates

The simpler case are invertible combiners, such as AGG_{sum}. The natural approach of evaluating multiple simultaneous windows would be to run multiple loop-fused instances of SoE. However, we found that the TwoStacks algorithm can be extended to support this case as well, yielding a more cache efficient approach. Somewhat surprisingly, this can be implemented using only two stacks (illustrated in Figure 2). Like the single query case, the elements of the back and front stacks share the same memory space and their aggregates are kept separately. Next, we will explain the algorithm and provide an example with two queries.

Algorithm 1: SlideSide (Inv) Pseudocode

```
Input: A set of aggregate queries Q, a combiner operation ⊕, an inverse operation ⊖
Output: The results of the window queries in Q
1 windowSize = Q.getMaxWindowSize()
2 backStack [windowSize+1] = {neutralVal} // used for prefix-scan
3 frontStack [windowSize+1] = {neutralVal} // used for suffix-scan
4 elements [windowSize] = {neutralVal} // used for input stream
5 curPos = 0
6 foreach val: stream do
7     insert(backStack, frontStack, elements, curPos, windowSize, val)
8     emitResults(backStack, frontStack, curPos, windowSize, Q)
```

Algorithm 2: Algorithm for insert(...)

```
1 // compute the suffix-scan
2 if (curPos==0) then
3     for i=0,1,..., windowSize do
4         frontStack[i+1] = frontStack[i] ⊕ partials[windowSize-i-1]
5 elements[curPos] = val
6 backStack[curPos+1] = elements[curPos] ⊕ backStack[curPos]
7 curPos = (curPos+1) % windowSize // wrap around the circular buffer
```

During the initialization phase of Algorithm 1, the *back* stack (blue row), the *front* stack (green row) and a circular buffer of *elements* (light-blue row) are allocated with size equal to the largest window from a given set of queries, Q, and initialized with the neutral element of the aggregate function (lines 1-4). For every input value *val* from the stream, we call the *insert* function and then compute the results for every query in Q with *emitResults* in line 6-8.

Upon the arrival of a new element, using the *insert* function (Algorithm 2), its *val* is stored in the next available slot in the circular buffer, defined by the *curPos* variable in line 5. The back stack is used for maintaining the prefix-scan of the input with every insert (line 6). If we reach the end of the elements buffer, we wrap around to the beginning and compute a suffix-scan over the input (lines 2-4) before applying the new insertion. Note that, as in TwoStacks, the computation of the suffix-scan occurs infrequently and the algorithm has $O(1)$ amortized complexity.

Algorithm 3: Algorithm for emitResults(...)

```
foreach query q : Q do
    curWindowSize = q.getSize();
    hasWrapped = false;
    endPtr = curPos;
    if (endPtr == 0) then
        endPtr = windowSize
    startPtr = endPtr - curWindowSize;
    if (startPtr < 0) then
        hasWrapped = true // the window wraps around the circular buffer;
        startPtr + = windowSize;
    if (!hasWrapped && startPtr == 0) then
        res = backStack[endPtr] // use the result from prefix-scan;
    else if (hasWrapped) then
        res = backStack[endPtr] ⊕ frontStack[windowSize - startPtr];
    else
        res = backStack[endPtr] ⊖ backStack[startPtr];
    forward answer res to query q;
```

After the insertion, the *emitResults* function (Algorithm 3) is called for computing the results for each query with the set

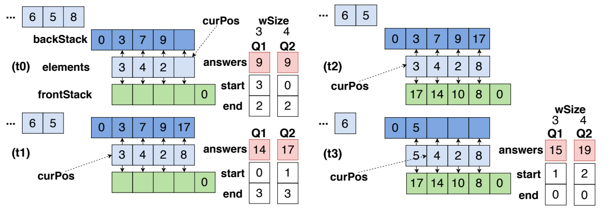


Figure 2: SlideSide (Inv)

Q. Based on the values of *curPos* and the window size of each query, this algorithm first computes the `start` and `end` pointers (lines 2-10) that are used for "bookkeeping". Next, we distinguish three different cases based on the value of these pointers: ***a)*** if the start pointer is 0, the result is already computed by the `prefix-scan` and returned in line 12; ***b)*** if the start pointer is greater than 0 and the end pointer does not wrap around the beginning of the elements buffer, the result value is computed by applying the *inverse* aggregate function on the values from the back stack in the positions of start and end pointers (line 16); ***c)*** if the end pointer has wrapped, the result is computed by combining `backStack[endPtr]` and `frontStack[winSize-startPtr]` in line 14.

In Figure 2, we present an example of SlideSide for two windows of size 3 ($Q1$) and 4 ($Q2$) and slide 1. The red boxes represent the result for the queries on each phase. The white boxes hold the values of the start and end pointers we discussed above. At our initial phase $t0$, the values 3, 4 and 2 have been already inserted in the *elements* buffer (from left to right) and their `prefix-scan` is computed in the back stack above (3, 7, 9). In the next phase $t1$, we have an insertion in the last slot of the *elements* buffer, which triggers the computation of the `prefix-scan` again for `backStack[4]` by combining the values 9 (previous result) and 8.

After the insertion, the algorithm is ready to emit results for both queries. For $Q2$, the result contains all the elements and the window starts from position 0 (case ***a)*** from above). Thus the result is already computed by the `prefix-scan` and can be obtained by accessing the value of `backStack[3]` which is 17. For $Q1$, the result contains only the latest three elements and the window points at positions 1 and 3 (case ***c)***). Thus, the result is computed by applying the *inverse* aggregate function on the elements from the back stack placed at that positions (17-3=14).

When we reach the end of the elements buffer ($t2$), we wrap around to the beginning and we compute a `suffix-scan` over the input using the front stack. In phase $t3$, we have the first eviction, where the latest input value 5 replaces value 3 and `backStack[1]` becomes equal to 5. Now, both query windows have wrapped (case ***b)***). This operation is going to return the `suffix-scan` of the remaining elements after evictions using the front stack and the current running aggregate from the back stack, which results in 15 and 19 respectively for queries $Q1$ and $Q2$.

3.2 Non-Invertible Aggregates

Processing multiple non-invertible functions can be performed with the Algorithm 1, but the `suffix-scans` have to be triggered more frequently. The algorithm is omitted to conserve space, but the logic is similar. The intuition behind this approach is that, while each of the queries maintains and operates on its own pair of stacks, these can be overlayed and start at the same memory address. In effect, the smallest stack is stored in the same memory region as the bottom part of the next larger, and so forth. To preserve correctness among the results, the computation of the `suffix-scan` is triggered every time the query with the smallest window size starts to evict.

In addition to the previous observations, we can apply optimizations proposed for single-query evaluation in HammerSlide [8], such as maintaining only the top value of the back stack. During the `suffix-scan` computation we can also stop propagating the changes from the current position until the end of the stack, if our computations do not alter the aggregate values, which reduces greatly the overhead of multiple flip phases and result in constant amortized complexity (see Table 1).

4 EVALUATION

In this section, we evaluate SlideSide for both invertible and non-invertible functions to show the benefits of our incremental strategy. To evaluate the efficiency of different aggregation algorithms, we run our experiments as a standalone prototype. We compare SlideSide to `SlickDeque` (for non-invertible functions we tradeoff performance with memory by using a fixed size deque), `TwoStacks` (using optimizations from [8]), `SoE` and `FlatFAT` when it's applicable (e.g., `SoE` is evaluated only for invertible functions). Each prototype maintains sliding windows with slide 1 by performing an eviction, an insertion and producing a result. We start our evaluation with the case we focus on: multi-queries and we demonstrate that SlideSide achieves higher performance. After that, we study the performance in the single query case, in which our solution exhibits only small performance loss.

4.1 Experimental setup and workloads

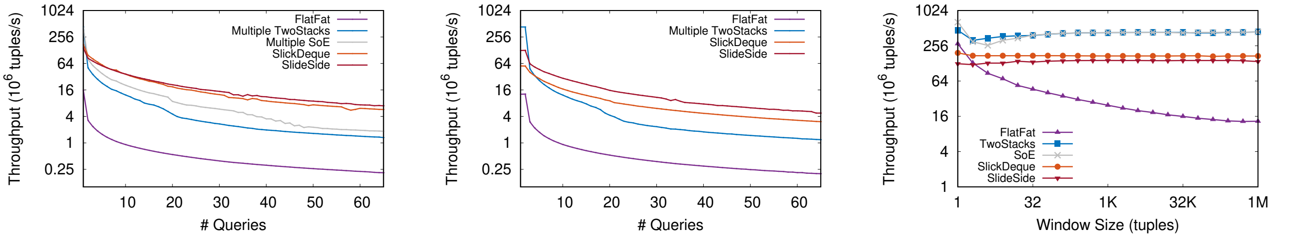
Hardware. All experiments are performed on a server with 2 Intel Xeon E5-2640 v3 2.60 GHz CPUs, a 20MB LLC cache and 64 GB of memory. We used Ubuntu 18.04 with 4.15.0-50-generic Linux kernel and compiled all experiments with clang++ version 9.0.0 and optimization level `-O3`.

Workload. Our workload emulates an anomaly detection scenario using the energy consumption trace from a smart electricity grid. This trace contains smart meter data from electrical devices in households [5] (32 bytes tuple size). We use two queries to perform analysis over the stream and detect outliers: SG_1, an aggregation that computes a sliding global AGG_{sum} and SG_2, which computes a sliding global AGG_{min} over the meter load.

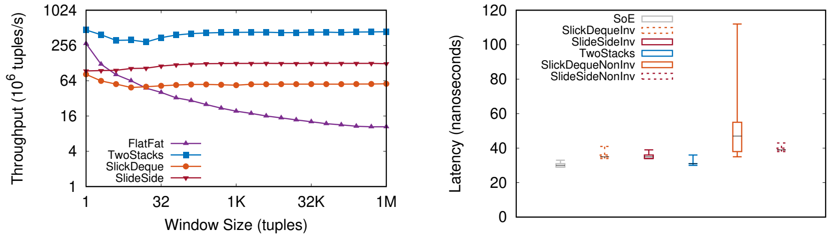
4.2 Multi-Query Evaluation

In the multi-query experiments, we generate queries of uniformly random window sizes (within the range [1, 32768] of tuples), while maintaining a constant window slide of 1 tuple for all of them. In this setup, we created workloads that contain from 1 up to 65 concurrent queries. `TwoStacks` and `SoE` can not be used to evaluate multiple queries, so we replicate their data-structures for every single window definition, as illustrated in Table 1.

Invertible Functions. For invertible functions, we are computing query SG_1 over different window definitions. Figure 3a demonstrates that `SoE` is the fastest algorithm and outperforms the multi-query solutions by up to 2.5× for a single query. However, as the number of queries increases, the overhead of maintaining multiple data-structure replicates becomes noticeable. Thus, we observe that the multi-query algorithms perform nearly 4×. Comparing SlideSide with `SlickDeque` reveals a small performance benefit that reaches up to 40% with the increase of query concurrency. Our approach allows the compiler to generate more efficient code, because of the simpler CPU instruction stream, while providing more predictable memory access.



(a) **Throughput of multi-queries** ($\mathrm{AGG}_{\mathrm{sum}}$) (b) **Throughput of multi-queries** ($\mathrm{AGG}_{\mathrm{min}}$) (c) **Throughput of single-query** ($\mathrm{AGG}_{\mathrm{sum}}$)



(d) **Throughput of single-query** ($\mathrm{AGG}_{\mathrm{min}}$) (e) **Latency in nanoseconds**

Figure 3: Comparison of incremental techniques

Non-Invertible Functions. For the non-invertible functions we are computing the $\mathrm{AGG}_{\mathrm{min}}$ over the generated windows. In Figure 3b, we observe that the multiple instances of `TwoStacks` outperform both SlideSide and `SlickDeque` for the first two and three workloads respectively. After that point, SlideSide is from 70% up to 2.2× faster compared to `SlickDeque` and more than 4× compared to the other two techniques. This illustrates that even though SlideSide requires more memory compared to `SlickDeque`, its CPU-cache-friendly data layout scales better with the number of queries in comparison to the deque data structure.

4.3 Performance Overhead for Single-Query

In this section, we present the efficiency of SlideSide for single-query workloads. We use queries $\mathrm{SG}_{1\text{-}2}$ to measure throughput and latency of the aforementioned approaches.

Throughput. For this experiment, we use SG_1 and SG_2 over windows with window sizes that vary between 1 and 1048576 tuples. Figure 3c illustrates the throughput penalty introduced by our algorithm for invertible functions in a single query scenario. SlideSide exhibits throughput nearly 3× worse than `SoE` and `TwoStacks`. In Figure 3d, we observe that `TwoStacks` is the best-performing non-invertible algorithm for different window sizes (440 million tuples/sec). In contrary, SlideSide is 3× worse but exhibits better performance than `SlickDeque`, because of its underlying data structure with sequential memory layout.

Latency. To measure the latency of all the previous approaches, we use a fixed window size of 32K tuple and window slide of 1. In Figure 3e, we omit the latency of `FlatFAT`, as it consistently is an order of magnitude higher than the other algorithms. We show that SlideSide exhibits latency that is comparable to the best-performing solutions for both invertible and non-invertible functions (minimal overhead) and better compared to the other multi-query solution, `SlickDeque`.

Overall, we observe that for single query evaluation SlideSide ends up exhibiting nearly 3× worse performance in throughput and similar latency compared to the best-performing approaches. This is the result of the memory pressure from maintaining extra dependencies (not needed by a single-query) along with a more complex CPU instruction stream that hinders optimizations.

5 CONCLUSION

In this paper, we presented a novel algorithm for highly efficient evaluation of multiple aggregate queries by maintaining a `prefix-` and a `suffix-scan` over the input. Our algorithm can be used as a drop-in replacement for any associative aggregation operator in a commercial streaming system, such as Flink [2] (e.g., as an aggregate store for Scotty[9]). SlideSide outperforms the state-of-the-art algorithms in multi-query scenarios by up to 2× in throughput, while exhibiting better latency. However, our study reveals that current window aggregation techniques do not exhibit robust performance across different types of aggregation functions and concurrency levels. Thus, a streaming engine will either perform poorly for different points within this design space or have to maintain multiple algorithms with a cost model.

REFERENCES

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z
[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™ : Stream and Batch Processing in a Single Engine. (2015). https://doi.org/10.1109/IC2EW.2016.56
[3] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. 1996. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proceedings of the Twelfth International Conference on Data Engineering*. 152–159. https://doi.org/10.1109/ICDE.1996.492099
[4] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17* (2017), 11–14. https://doi.org/10.1145/3093742.3095107
[5] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 266–269. https://doi.org/10.1145/2611286.2611333
[6] Anatoli U Shein, Panos K Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. Section 4 (2018), 397–408. https://doi.org/10.5441/002/edbt.2018.35
[7] Kanat Tangwongsan and Martin Hirzel. 2015. General Incremental Sliding-Window Aggregation. *Pvldb* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940
[8] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2018. Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation, See [8], 34–41. http://www.adms-conf.org/2018-camera-ready/SIMDWindowPaper_ADMS%2718.pdf
[9] J. Traub, P. M. Grulich, A. Rodriguez Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. 2018. Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1300–1303. https://doi.org/10.1109/ICDE.2018.00135