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 TwoStacks, 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).

© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.
Algorithm Time Space

Single Query Multi-Queries Single-Query Multi-Queries Amort Worst Amort Worst

\[ \text{SoE}[^4] \quad \text{Inv} \quad 2 \quad 2 \quad q \quad q \] 2 \times 2 \times 2 \times 2

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

SoE[^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 pushed 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 flips 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 \( O(\log n) \) 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 \( AGG_{\text{max}} \) 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 \( AGG_{\text{sum}} \). 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.

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 1: SLIDESIDE (INV) Pseudocode

```plaintext
1 // compute the suffix-scan
2 if (curPos == 0) then
3   for c = 1 to windowSize do
4     frontStack[c+1] = frontStack[c] ⊕ partials[windowSize-i-1]
5     elements[curPos] = val
6     if (curPos == windowSize) then
7       backStack[curPos] = elements[curPos]
8     else
9       backStack[curPos] = backStack[curPos] ⊕ val
10     endPos = (curPos-1) % windowSize // wrap around the circular buffer
```

Algorithm 3: Algorithm for emitResults(...)

```plaintext
1 foreach query q ∈ Q do
2   curWindowSize = q.getWindowSize();
3   hasWrapped = false;
4   endPtr = curPos;
5   if (endPtr == 0) then
6     endPtr = curWindowSize;
7     startPtr = endPtr - curWindowSize;
8     if (startPtr < 0) then
9       hasWrapped = true // the window wraps around the circular buffer;
10      startPtr += curWindowSize;
11     if (hasWrapped && startPtr == 0) then
12       res = backStack[endPtr] // use the result from prefix-scan;
13     else if (hasWrapped) then
14       res = backStack[endPtr] ⊕ frontStack[windowSize - startPtr];
15     else
16       res = backStack[endPtr] ⊕ backStack[startPtr];
17     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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Single Query</th>
<th>Multi-Queries</th>
<th>Single-Query</th>
<th>Multi-Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoE[^4]</td>
<td>( n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>FlatFat[^7]</td>
<td>( \frac{1}{2}n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>SlickDeque[^6]</td>
<td>( \frac{1}{2}n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>SlickDeque</td>
<td>( \frac{1}{2}n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>TwoStacks</td>
<td>( n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>SlickDeque</td>
<td>( \frac{1}{2}n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>TwoStacks</td>
<td>( n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
<tr>
<td>SlickDeque</td>
<td>( \frac{1}{2}n )</td>
<td>( 2 \times q )</td>
<td>( q )</td>
<td>( 2 \times q )</td>
</tr>
</tbody>
</table>

```
In addition to the previous observations, we can apply optimizations proposed for single-query evaluation in Hammer-Side [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 SlickDeque 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 SlickDeque 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 SlickDeque 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, an aggregation that computes a sliding global AGGsum and SG, which computes a sliding global AGGmin 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 cannot 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 SG1 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 SlickDeque 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.
Non-Invertible Functions. For the non-invertible functions we are computing the AGG\textsubscript{inv} over the generated windows. In Figure 3b, we observe that the multiple instances of TwoStacks outperform both SlickDeque and SlickDeque\textsubscript{non} for the first two and three workloads respectively. After that point, SlickDeque\textsubscript{inv} 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 SlickDeque\textsubscript{inv} 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 SlickDeque for single-query workloads. We use queries SG\textsubscript{1-2} to measure throughput and latency of the aforementioned approaches.

Throughput. For this experiment, we use SG\textsubscript{1} and SG\textsubscript{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. SlickDeque\textsubscript{inv} exhibits throughput nearly 3× worse than SG\textsubscript{2} 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, SlickDeque\textsubscript{inv} 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 SlickDeque\textsubscript{inv} 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\textsubscript{non}.

Overall, we observe that for single query evaluation SlickDeque\textsubscript{inv} 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]). SlickDeque\textsubscript{inv} 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

Figure 3: Comparison of incremental techniques

(a) Throughput of multi-queries (AGG\textsubscript{sum})  (b) Throughput of multi-queries (AGG\textsubscript{min})  (c) Throughput of single-query (AGG\textsubscript{sum})
(d) Throughput of single-query (AGG\textsubscript{min})  (e) Latency in nanoseconds

438