Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation

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

Alexandros Koliousis
Imperial College London
akolious@imperial.ac.uk

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

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

ABSTRACT

The computation of sliding window aggregates is one of the core functionalities of stream processing systems. Presently, there are two classes of approaches to evaluating them. The first is non-incremental, i.e., every window is evaluated in isolation even if overlapping windows provide opportunities for work-sharing. While not algorithmically efficient, this class of algorithm is usually very CPU efficient. The other approach is incremental: to the amount possible, the result of one window evaluation is used to help with the evaluation of the next window. While algorithmically efficient, the inherent control-dependencies in the CPU instruction stream make this highly CPU inefficient.

In this paper, we analyse the state of the art in efficient incremental window processing and extend the fastest known algorithm, the Two-Stacks approach with known as well as novel optimisations. These include SIMD-parallel processing, internal data structure decomposition and data minimalism. We find that, thus optimised, our incremental window aggregation algorithm outperforms the state-of-the-art incremental algorithm by up to $11\times$. In addition, it is at least competitive and often significantly (up to 80%) faster than a non-incremental algorithm. Consequently, stream processing systems can use our proposed algorithm to cover incremental as well as non-incremental aggregation resulting in systems that are simpler as well as faster.

1. INTRODUCTION

Stream processing has applications ranging from credit card fraud detection [8] to click stream analytics [9, 7, 2]. A challenge in these applications is to evaluate complex queries over a continuous stream of input tuples (e.g. credit card transactions, or click logs) at high throughput and low latency. One of the key operations applied in such stream queries is window aggregation [3], i.e. the calculation of running aggregates in a FIFO buffer [2, 21]. In addition, window aggregation is used for operations on persistent data such as time-series computation [18, 24].

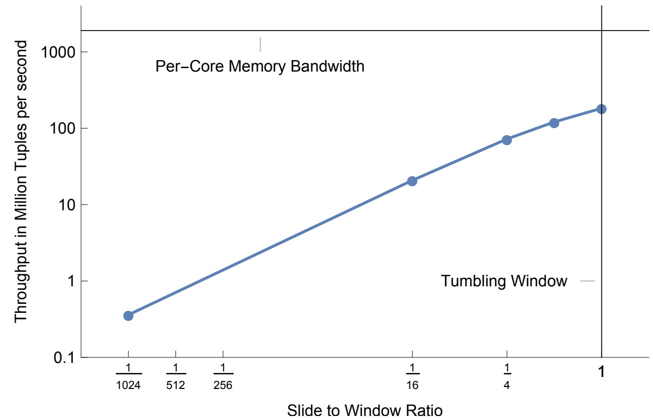


Figure 1: Tumbling window aggregation is easy to optimise; sliding window aggregation is not.

Formally, window aggregation forms a sequence of finite subsets of a (possibly infinite) input dataset and calculates an aggregate for each of these subsets. We refer to the rules for generating these subsets as the *window definition*: its *size*, i.e. the number of data items to be included in an aggregate and its *slide*, i.e. the "distance" between the first tuple in the current window and the first tuple in the next window. A window whose slide is equal to its size is called *tumbling*; if the slide is less, it is called *sliding*. If the slide is greater than the window, it is *sampling*—which we will consider out of scope for this paper. We focus on both count-based and time-based windows of fixed size and assume a streaming setup, i.e. only consider single-pass algorithms with bounded memory requirements. We also limit ourselves to distributive and algebraic aggregation functions (min/max, sum, average, ...) and exclude holistic ones (like percentiles).

While conceptually similar, tumbling and sliding windows are usually distinguished by their opportunities for intermediate result sharing: in tumbling windows, every input tuple contributes to exactly one window aggregate; in sliding windows, a tuple contributes to more than one. In other words, the amount of necessary work per input tuple is bounded (upwards) by the maximum number of open windows. In the worst case, i.e. when the slide is one, this is equal to the window size. As a result, the performance of sliding window queries is typically dominated by factors other than the memory bandwidth, which distinguishes them from tumbling windows and classic relational aggregation.

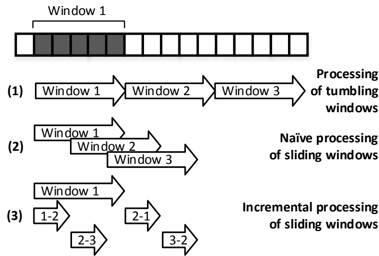
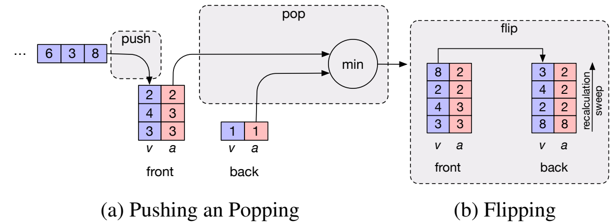


Figure 2: The case for incremental window computation



(a) Pushing an Popping (b) Flipping

Figure 3: The Two-Stacks algorithm

To illustrate this, we implement a single-threaded running average query over a window of 1024 integers and vary the slide from 1 to 1024 (at which point the query turns into a tumbling window).[1] Figure 1 shows that, for the tumbling window, the query saturates approximately a fifth of the per-core bandwidth. There is thus potential to improve performance with a SIMD version of the query—as tumbling windows are equivalent to relational aggregation, the potential is well documented [10, 20]. More interestingly, however, the gap between memory and processing bandwidth is more striking in the sliding window case: there is an imbalance of more than three orders of magnitude.

Fortunately, there are many opportunities for intermediate result sharing in sliding window aggregation using *incremental* algorithms and data structures [5, 24, 25, 11]. Considering our previous example of a running average query, an incremental algorithm does not recompute the average over 1024 values each time the window slides. It rather adds the new values that entered the window and subtracts those that fell out of it. Incremental algorithms, however, commonly expose many control and data dependencies in the CPU instruction stream. This, in turn, hinders opportunities for efficient superscalar execution and SIMD parallelism, reducing potential performance gains.

In this paper, we describe how to achieve significantly better performance for state-of-the-art incremental window-aggregation algorithms through careful, hardware-conscious optimisation. We make the following contributions:

- We analyse the performance of the three best-performing window-aggregation algorithms from recent literature [11]. We show how all of them – and arguably the entire problem of streaming window aggregation – expose fundamentally different performance patterns than classic database problems: they tend to suffer much less from memory bandwidth starvation and much more from L1-data cache latency and control-hazards.

- Based on that observation, we study the applicability of a number of optimisation techniques, including the design and implementation of vectorized operators, to address the identified bottlenecks. Furthermore, we demonstrate how each one of these optimisations affects the runtime of our implementation.

- Finally, we demonstrate that the best-performing incremental algorithm for sliding window computation [1, 11], can, if carefully optimised, be competitive with a highly optimised non-incremental algorithm, even when processing tumbling windows. Besides the potential to simplify stream processing systems, this also yields an implementation that is almost 2× faster than either competitor when the slide is greater than one but less than the window size.

[1] We implement in C++, compiled with `gcc` using `-O3 -mtune=skylake`.

The rest of this paper is organised as follows: we survey the state-of-the-art in incremental window computation in Section 2. Then, we conduct a hardware-conscious evaluation of the incremental algorithms under study and, based on our findings, we develop a number of optimisations in Section 3. We evaluate our optimisations and present advise on how to select the most appropriate aggregation algorithm in Section 4. Finally, after presenting related work in Section 5, we discuss future work and conclude in Section 6.

2. BACKGROUND

Next, we provide background on window aggregation and the best performing incremental algorithms for sliding windows.

2.1 Parallel Window Aggregation

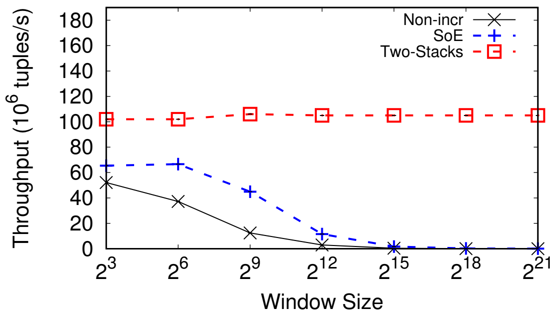
As mentioned above, we support both count- and time-based windows by statically pre-allocating our data structures. We define n as the window *size* and ℓ as the *slide*. A new window starts after each ℓ tuples in the system and ends n tuples later, allowing for the possibility of multiple open windows.

Unlike other stream operators, such as selection or projection, window aggregation is a *stateful* operator. To produce an output tuple, all of the tuples in the window must arrive and be buffered. It is that property that makes it hard to parallelise the window aggregation operator itself. A common approach is to parallelise *on top* of the aggregation operator, i.e. process multiple windows in parallel. We show this process in Figure 2: when processing tumbling windows in case (1), Windows 1, 2 and 3 can be processed independently without redundant work. In case (2), the sliding windows are processed independently but with redundant work: tuple 6, e.g. contributes to windows 1, 2 and 3 and thus is processed three times. In the last case (3), the stream is broken down into fixed-size chunks, commonly referred to as *panes* [16]. Panes are processed independently and in parallel. After all contributing panes are processed, the final window result is calculated from the result of the processed panes. Most popular multi-core stream processing systems, e.g. SABER [14], Spark Streaming [27] and Trill [7], process data in this manner.

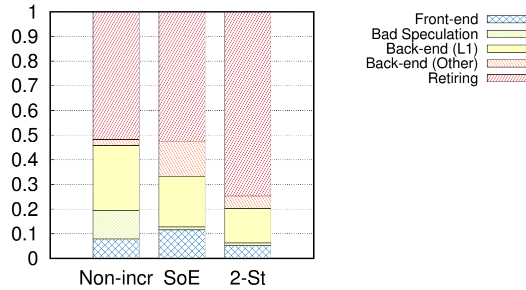
In this paper, we focus on the processing that occurs *within* a pane—the work-efficient production of an incremental result. Our work naturally complements existing research on the problem of parallel stream processing.

2.2 Incremental Window Computation

Depending on the aggregation function, different window aggregation algorithms are appropriate. Following previous convention [11], we distinguish “sum-like” (e.g. sum, count, or average) and “min-like” (e.g. min or max) aggregation. Sum-like functions are *invertible*, i.e. the value of window $n+1$ can be calculated from the value of window n and the difference between the two window inputs (the newly entered and evicted tuples). This avoids the need



(a) $\mathsf{AGG}_{\mathsf{min}}$ with varying window sizes and slide 1



(b) Cost Breakdown by CPU components for window size 1024

Figure 4: $\mathsf{AGG}_{\mathsf{min}}$ with slide 1

to scan all tuples in the window at each step. However, for non-invertible aggregations, this does not hold. For instance, with min aggregation, when the current minimum leaves the window, the entire window must be scanned to find the new running value. Note that, when evicting a value greater than the current minimum, the scan can be avoided.

Based on a recent survey of stream processing algorithms [11], we select the best-performing algorithms. All have $O(1)$ insertion time per tuple, and $O(n)$ space complexity. The window computation time ranges from $O(\ell)$ to $O(n)$.

- *Non-incremental.* The non-incremental algorithm recalculates the aggregate result for each window from scratch. Its time per-tuple complexity is $O(n)$.

- *Subtract-on-Evict (SoE).* This algorithm re-uses the previous window result to compute the current one by removing the expired values and merging in the added ones. As discussed, for non-invertible aggregates, it may still cause a rescan of the window.

- *Two-Stacks.* This algorithm [1], shown in Figure 3 for the case of minimum, maintains two stacks, *front* and *back*, to store the window: when a new value v arrives, it is **push**ed onto the front stack. Besides the values themselves, each stack maintains a second column with the following invariant: the aggregate a at position p from the bottom of the stack is the aggregate of all values v at positions 1 through p. Figure 3 shows that in the case of computing the $\mathsf{AGG}_{\mathsf{min}}$ the red column maintains the aggregates at each position incrementally with every insert, while in the blue column we store the respective value. The **pop** operation removes the top from the back stack and aggregates it with the top of the front stack. If the back stack is empty, the algorithm **flip**s the front onto the back, reversing the order of the values and recalculating the aggregates in $O(n)$. However, as the flip phase occurs infrequently, Two-Stacks achieves $O(1)$ amortised time complexity.

While other tree-based algorithms [24] can support more general aggregate functions, such as median or percentiles, we omit them because they have an even greater maintenance overhead.

3. OPTIMISING WINDOW AGGREGATION

Let us now describe our efforts on optimising incremental window aggregation algorithms. We start with a CPU-conscious analysis of the algorithms followed by a description of our optimisations.

3.1 Analysis

Having introduced the three basic algorithms, we measure their absolute performance, after exploiting obvious opportunities for optimisation. We evaluate them in the most challenging case: a min aggregate (non-invertible) over a count-based window of slide 1. We generate a stream of random integers, uniformly distributed in [0,64), and vary the size from 8 to 2,097,152 (see Section 4.1 for details of our hardware set-up).

Figure 4a shows the results. We observe that the Two-Stacks algorithm is affected little by the window size; the other two algorithms experience severe performance degradation as the window size increases. For the SoE algorithm, this is expected: the cost of computing the new minimum if the current value is evicted increases with the window size. The computation is performed by re-scanning the entire window. The decreasing performance of the naïve, non-incremental algorithm is consistent with the explanation in Section 1. In their unoptimized form, the Two-Stacks algorithm is thus the most efficient. However, it is unclear how efficient each of these algorithms is from a microarchitecture view.

To answer that question, we break down the elapsed cycles by CPU component, as described in [12]. Figure 4b shows that all algorithms spend at least 50% of their cycles retiring instructions. In most of the remaining cycles, the execution is back-end bound, due to L1 cache accesses. This is consistent with our expectations since most of the data accesses are performed to manage the window state which resides in L1 cache. Overall, we found that the best performing algorithm, i.e., Two-Stacks, spends less than 10% on bad speculation or front-end bound. This indicates that there are two opportunities for performance improvement: (i) improving CPU efficiency; and (ii) reducing the number of L1 data accesses. Next, we discuss optimisations to both ends.

3.2 Optimisations

In this section, we discuss a number of optimisations, which will help us to address the bottlenecks that we identified in Section 3.1. In order to support our design decisions and evaluate their importance, we provide a graph at the end of the sub-sections to show the effect of each optimisation on the runtime. For that purpose, we compute the min for windows of size 1024 and slide 64.

(1) Storing data in ring buffers. Our first optimisation concerns both the Two-Stacks and SoE algorithms. A generic implementation can allow the state of these algorithms to grow arbitrarily, offering the flexibility to process windows that vary in size (e.g. time-based windows). This flexibility comes at the cost of more complex addressing logic and bounds checking (indexed access must perform two pointer dereferences). We avoid these overheads by us-

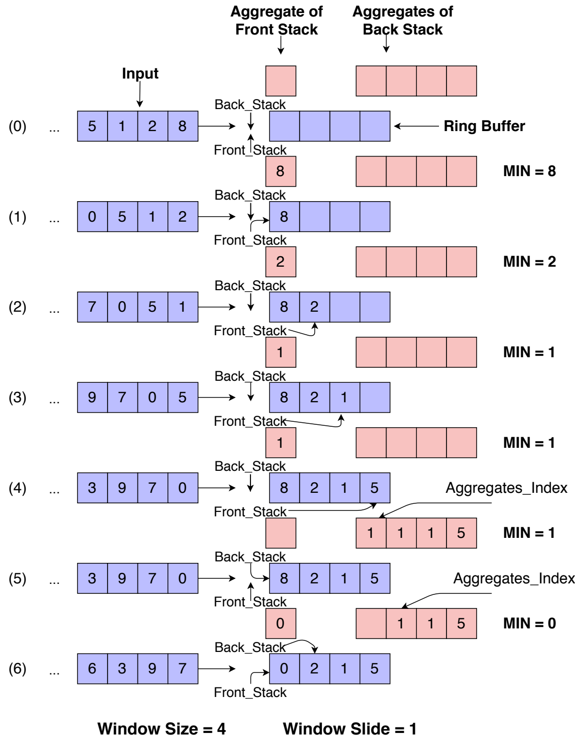


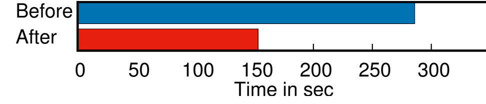
Figure 5: Two-Stacks Implementation with Ring Buffer

ing ring buffers, allocating sufficient space to fit the entire window in our experiments. The logic of ring buffers allows us to process unbounded streams of data, by wrapping around to the beginning of the underlying allocated memory, and offer a predictable access pattern. At a hardware level, this data layout is CPU-cache-friendly because tuples can be pre-loaded. Below, we evaluate our structure with stacks from the C++ Standard Template Library (STL), that use deque as their standard container and require two pointer dereferences.

We implement the ring buffers as fixed-size arrays and a modulus-divide on the access cursors. Storing such a "flat" sequence of tuples in contiguous memory regions enables optimisations, such the use of SIMD instructions. Thus, we decide to use these ring buffers for all our data structures, both queues and stacks, because of their simplicity and high efficiency.

In Figure 5, we present the implementation of the Two-Stacks algorithm with a flat array in the case of min aggregate, for a window of size 4 and slide 1. The front and back stacks are defined by the respective pointers of our structure, which we move according to the window slide and size. For each phase, apart from the ring buffer, we utilise and maintain a variable and an array for the running aggregate values of the front and the back stack respectively. The tuples enter the system in the order: 8, 2, 1, 5, 0, 7, 9, 3, 6.

In phase (1), the front stack has already one element, while the back stack is empty. Next, in phases (2), (3) and (4) we insert new values, by moving the front stack's pointer and changing the aggregation value, if necessary. At this point, our window has four elements and we are ready to emit the result, but the back stack is still empty. Thus, we have to perform the flip phase (5) of the algorithm. During this phase, we do not have to copy any of the intermediate data from the front to the back stack. Instead, we traverse the array backwards to recalculate the aggregates. This optimisation removes the unnecessary writes in the flip phase. Moreover, in combination with the data locality of the contiguous memory, they improve significantly the performance reducing the runtime from 286s to 153.5s.

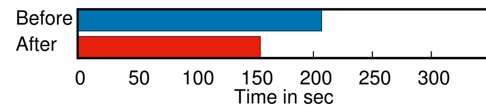


Finally, in phase (6), we evict the first element from the back stack and make an insertion to the front stack. When an eviction is triggered, we keep track of the valid aggregates of the back stack and remove the ones that do not contribute to the current result, by moving the Aggregates_Index pointer.

(2) Decomposed intermediates & Data minimalism. The Two-Stacks algorithm maintains two attributes in the window state: the inserted value and the aggregate of all the values beneath it. It is a fact that these can be stored in n-ary form (also known as struct-of-arrays or, more colloquially, columnar format). While decomposition does not yield an immediate performance benefit (as the application is not memory-bandwidth bound), it enables a number of subsequent optimisations, which we discuss in the following.

First, we observe that the front and the back stack are used differently (see Figure 3). From the front stack, only the value column and the top element of the aggregate column are ever read. The value column is only needed to perform the flip. From the back stack, only the aggregates are read by the **pop** operation. By exploiting the decomposed format of the stacks, we can thus elide the allocation and maintenance of the respective data buffers: instead, we only store the value column and the top element of the aggregate stack from the front stack and only the aggregates from the back stack. This reduces the number of L1 cache misses since fewer attributes have to be stored in cache.

Based on these observations, we implement the Two-Stacks algorithm as shown in Figure 5 and maintain the least possible values that allow us to compute the aggregation result. For window slides greater than 1, we realise that it's sufficient to maintain one single value per window slide for the back stack, as all the elements of a specific slide will be evicted at the same time. These optimisations yield an effective reduction of the runtime from 207.5s to 155.7s.



(3) Bulk insertion & SIMD scanning. To reduce the computational cost, it is natural to use vector instructions for the insertion of new values into the respective data structures. We do so by inserting the input values in bulk. For slides greater than one, which imply reduced output granularity, we also pre-aggregate the values upon insert. Since we have replaced the stacks with ring buffers in the Two-Stacks approach, this optimisation naturally applies to both algorithms. For example, in the insertion phases like (2), (3), (4) and (6) of Figure 5, if the slide is large enough, we can apply our aggregation functions with AVX2 intrinsics on each slide, to compute the running value of the front stack.

The main benefit of decomposed storage is that it enables the use of SIMD instructions for scanning the buffers when calculating aggregates. We implement a SIMD-parallel version for every aggregation function with AVX2 compiler intrinsics for both the SoE and Two-Stacks approaches on top of ring buffers. More precisely, in the case of the SoE algorithm, we apply these parallel versions during the eviction phase, when we want to recompute the

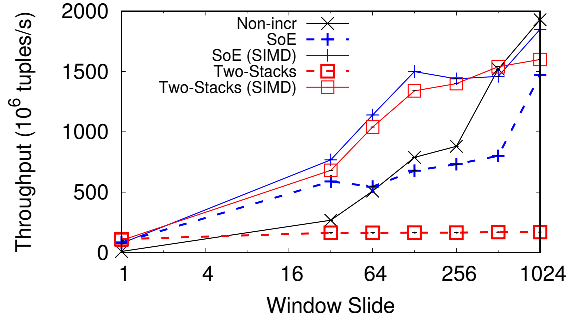


Figure 6: AGG_{avg} with window size 1024 and variable slide

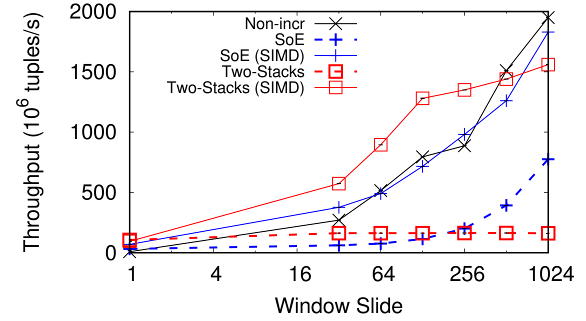
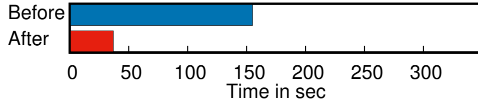


Figure 7: AGG_{min} with window size 1024 and variable slide

running value of the aggregation. For the Two-Stacks approach, we utilise this optimisation within the flip phase, as we show in (5) of Figure 5, in which we compute a single aggregate value per slide. These optimised functions for scanning the buffers reduce the runtime from 155.6s to 37s.



One may assume that the use of SIMD is not contingent on decomposed storage because shuffle or gather instructions may be used to arrange the values of a column in a contiguous memory region. We find, however, that the overhead of those instructions is both high and avoidable, by using decomposed storage.

Both SIMD scanning and the bulk insertion are applicable to aggregation functions, other than the ones we implement in this work. Their combination introduces parallel processing within a pane for single-threaded computation and is the most important improvement factor among the optimisations we apply.

(4) Stack RLE compression. Referring again to Figure 3b, we notice that the aggregate values on the back stack[2] expose an exploitable pattern: since each slot contains the minimum of the slot beneath it and its aligned value, the probability is high that the value in a slot is equal or smaller to the value in the slot beneath it. This data pattern is naturally amenable to run-length encoding. Instead of inserting a new value and increasing the top pointer, only the top count needs to be increased. This not only reduces the size of the stack but also allows us to do fast comparisons and scan the front-stack using SIMD instructions.

Unfortunately, we found that while this optimization reduces the number of L1 cache accesses, it introduces additional data-dependencies in the stack flipping code and as a result yields a performance reduction between 9 and 18 percent.

4. EVALUATION

To evaluate the relative merits of the existing algorithms as well as our optimisations we conduct experiments on synthetic as well as real-world datasets. In addition to the count-based windows that are the focus of our paper we also present a brief evaluation of how our optimisations apply to time-based windows as long as the maximum size of the window can be bounded.

For brevity, in this section, we only discuss all our optimisations in combination and do not evaluate the specific contribution of each optimisation for every use case. We hope to be given the opportunity to study this matter in more detail in an extended version of this paper.

4.1 Experimental Set-up

Hardware. All experiments are performed on a server with 2 Intel Xeon E5-2640 v3 2.60 GHz CPUs with a total of 16 physical CPU cores, a 20MB LLC cache and 64 GB of memory. We used Ubuntu 14.04, a 4.4.0-116-generic Linux kernel and gcc version 7.3.0. We decided to test all the algorithms with the minimum NUMA interference by binding our experiments to processor 0 (on NUMA node 0). The same applies to all our memory allocations, which were bound on that NUMA node.

Datasets and queries.

For our microbenchmark evaluation, we generate data streams of 32-bit integer values drawn from a uniform distribution. On this input dataset, we evaluate minimum and average aggregates, which are representative of the two classes of functions we study. We identify two separate experiments: 1) we keep the slide constant at 1 while changing the window size and 2) we keep the window constant at 1024 and alter the window slide, starting from slide 1 until our window becomes tumbling at 1024.

For the macro-evaluation we study a workload that emulates a cluster management scenario. The dataset represents a trace of time-stamped measurements taken from an 11,000-machine compute cluster at Google [26]. Each tuple contains information about metrics related to tasks executed on the cluster, such as cpu utilisation or task's priority. On that dataset, we evaluate a query that expresses a common cluster monitoring task [13] and reports the average requested CPU utilisation of submitted tasks.

4.2 The Effect of Window Slide

To study the effect of the slide on the throughput of the different implementations, we study average- and minimum-aggregation in Figures 6 and 7, respectively.

For both aggregation functions, we observe that the plain Two-Stacks algorithm performs robustly but almost across the board worst of all our studied implementations. For the average-aggregation the plain SoE is quite competitive and outperforms the non-incremental algorithm for slides less or equal to 64. However, we find that the SIMD-optimised versions of both algorithms perform significantly better than their unoptimised counterparts (up to 11× better in the case of the Two-Stacks algorithm). For minimum-aggregation, the SIMD-enabled Two-Stacks algorithm outperforms all others (by almost 80%) best for slides less than half of the window size and is never more than 10% worse than the best algorithm. Given the intricate nature of the implementation and the fact that there is no

[2] Technically, the front-stack as well, but we have eliminated the aggregates from it.

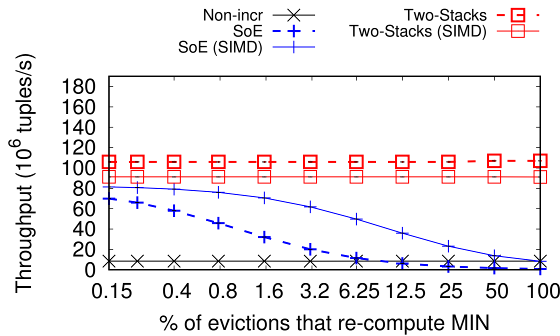


Figure 8: How the evictions influence the throughput – AGG_{min} with window size 1024 and slide 1.

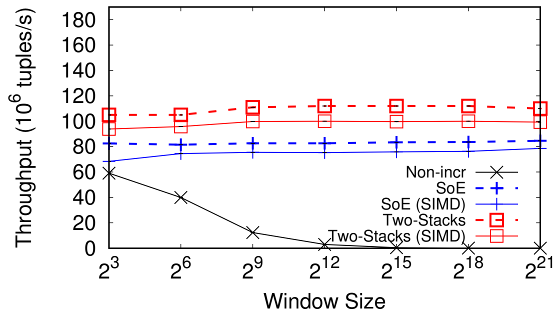


Figure 9: AGG_{avg} with variable window size and slide 1

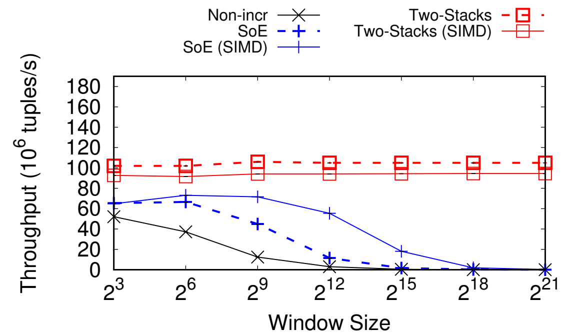


Figure 10: AGG_{min} with variable window size and slide 1

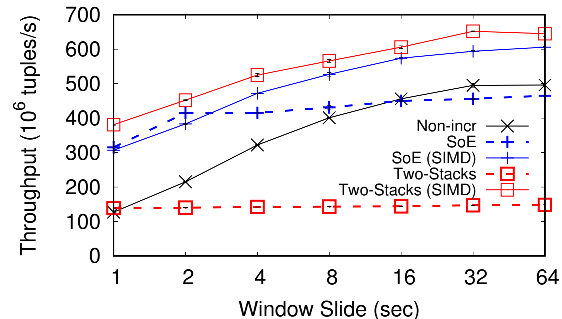


Figure 11: AGG_{avg} with time-based windows of size 64 seconds and variable slide over the Google cluster stream

potential for re-use to exploit one might expect worse behaviour due to the complex CPU instruction stream. It turns out, however, that our careful optimisation yielded an implementation with very little overhead.

4.3 The Effect of Window Re-computations

When comparing the SoE to the Two-Stacks approach for min-aggregation, the advantage of the Two-Stacks lies in the robustness against adversarial input distributions: in the worst case every eviction causes a re-computation of the window. To quantify that problem we study performance degradation of the different algorithms with the rescan rate (i.e., the percentage of evictions that cause a re-compute). For that purpose, we set the window size to 1024 and the slide to 1. Further, we restrict the number of unique values in our input distribution to force more re-computations and present the results in Figure 8. We observe that both SoE implementations experience severe performance degradation when the re-compute rate increases. The figure shows that all Two-Stacks implementations are robust against that input distribution.

4.4 The Effect of Window Size

Next, we vary the size of a count-based window with a constant slide of 1 and consider again two aggregates, AGG_{avg} (Figure 9) and AGG_{min} (Figure 10). In both cases, for all window sizes, the two-stacks algorithm has the highest throughput at approximately 110 million tuples/s. Since the slide is 1, the optimised version of two-stacks is slightly slower because of the overhead of checking for opportunities to vectorise code (but in this case, there are none).

For AGG_{min} (Figure 10), the throughput of two-stacks is not affected by the frequent evictions of the minimum value of the current window, as the algorithms can recompute the result in constant time. Note that the percentage of inserts that evict the current minimum is 6.25%. All other algorithms are affected by the growth of the window size: the overhead of recomputing the minimum from scratch over the entire window dominates and causes a significant drop in throughput, up to $100\times$. Only the optimised SoE algorithm can cope with that overhead, achieving only a $2\times$ slow-down compared to two-stacks when the window size is less than 4096 tuples because it benefits from vectorised instructions to re-scan the window.

For AGG_{avg} (Figure 9), the superior performance of Two-Stacks over SoE is less pronounced: it is only $1.5\times$ faster. The non-incremental algorithm experiences a similar fall in throughput as in the case of AGG_{min}. The non-incremental algorithm can be up to $300\times$ slower than Two-Stacks when the window size is 32,768.

4.5 Time-Based Queries on Real-World Data

Next, let us briefly examine how the presented optimizations apply to time-based window queries. For that purpose, we evaluate a straight-forward extension of our approach that uses an a priori known bound on the window size is but is otherwise fully functional and equivalent to the presented approaches.

For this part of the evaluation, we use the Google cluster trace. Noteworthy is that the load exposed by the trace is very spiky: the event-rate varies from 0 to 100K events per second. It thus, significantly stresses our implementations' ability to deal with varying window sizes.

On that dataset we evaluate AGG_{avg} with a time-based window of 64 seconds and a variable window slide from 1 to 64 seconds.

Figure 11 shows that the plain Two-Stacks algorithm does not perform well on time-based windows: its throughput, steady at roughly 150 million tuples/s for all slides, is hindered by frequent,

unoptimized flipping the front stack to the back. The non-incremental algorithm, on the other hand, outperforms Two-Stacks when the slide is greater than 1 second and matches the throughput of the SoE approach when the slide is 16 seconds or higher. This can be explained by the fact that slides from 16 onwards contain at least half of the window data because the input data distribution is skewed, thus recomputing a window result from scratch imposes no extra computation overhead—in fact, it is even cheaper than maintaining state for the SoE algorithm. Both optimised versions of Two-Stacks and SoE perform best, achieving a speed-up between $1.2 - 4.3\times$. These benefits are not just due to vectorised instructions: the re-design of the flip operation in Two-Stacks. Similarly, the SoE algorithm benefitted from avoiding redundant memory copies.

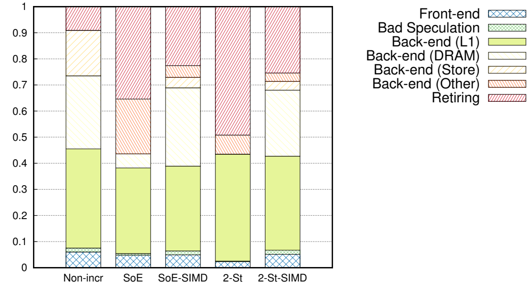


Figure 12: Cost Breakdown by CPU components for a tumbling window of size 1024.

4.6 Analysis & Advice

Before concluding, let us briefly discuss a final performance analysis: the by-CPU-component breakdown of all presented approaches for tumbling windows. This breakdown allows us to assess if there is still untapped performance potential in any of the implementations.

By analyzing the cost components of tumbling windows with size 1024, we want to understand the difference in performance between all the algorithms. In Figure 12, we witness that the best performing approaches (naive & vectorized) become nearly 30% DRAM bounded, which indicates that they approach the main memory bandwidth limits. This is consistent with our observations in Figure 7 in which the tumbling window throughput reached almost 2 billion tuples (or 8 Gigabytes) per second.

Overall our conclusion from Figure 12 is that our optimizations have transformed the performance profile of the plain SoE and Two-Stacks algorithms to be closer to that of the non-incremental algorithm: strongly dominated by memory-bandwidth but only little by computational resources (Retiring Instructions). Thus, our optimized approaches bridge the gap between sliding and tumbling windows and presents a significant improvement in terms of throughput compared to the rest, with more than $1.6\times$ greater throughput.

We have to note that even though we do not consider latency as a performance measure in the above experiments, both Two-Stacks implementations exhibit average latency lower than 15 nanoseconds. Even though they maintain such a low average latency, we observe periodic latency spikes caused by the flip phase, as we expected. Finally, the SoE algorithm has comparable latency in the case of invertible functions. For non-invertible functions, the latency is affected by the number of evictions (see Section 4.3).

5. RELATED WORK

DABA [25] is an alternative to Two-Stacks that ensures lower standard deviation in latency compared to the original algorithm. Each time the front stack is empty, the Two-Stacks algorithm has to perform a flip from the back stack onto the front stack. This high-cost operation introduces latency spikes. DABA, instead, gradually flips the back stack to the front stack with every push and pop operation. Thus it removes this heavy step and reduces the latency spikes, with a small throughput overhead.

FlatFIT [22], and its successor SlickDeque [23] were designed to support efficient sharing between different windows on the same stream simultaneously. In this work, we do not focus on multi-query execution. Fundamentally, the approach of SlickDeque for invertible functions considering a single query is equivalent to the SoE algorithm. When it comes to non-invertible functions in a single query, the approach of maintaining the positions of current partial aggregates is orthogonal to our solution of scanning the buffers with AVX2 compiler intrinsics.

Incremental computation for holistic aggregation operators (e.g. median-like functions) requires a tree-based data structure. The B-Int [4] algorithm supports multi-query execution with a B-tree, sharing the intermediate results of small-time overlapping intervals among multiple windows. FlatFAT [24] is a pre-allocated pointer-less tree implementation that, in contrast to the previous approaches, does not require FIFO windows (stream data items are not ordered by timestamp). We omit these tree-based approaches because we focus on distributive and algebraic aggregation functions.

In addition, Lies et al. [15] proposed a general algorithm for a window operator, in order to efficiently partition the computation per key across multiple cores, along with a specialized data structure, the Segment Tree. This structure enables the parallel computation of aggregates. Our work is about optimising the computation of a single key in a core.

6. SUMMARY AND FUTURE WORK

In this paper, we studied the efficient implementation of streaming window aggregation. We conducted an in-depth study of the best performing algorithms for sliding as well as tumbling windows and concluded that incremental algorithms, while work efficient, are highly CPU-inefficient. To address that problem, we developed a set of optimization techniques, some known, some new, and applied them to the two fastest incremental aggregation algorithms, i.e., *Subtract-on-Evict* and *Two-Stacks*. The result is incremental window-aggregation algorithms that are highly CPU-efficient and introduce parallel processing within a pane. In fact, they are competitive with non-incremental algorithms for tumbling windows and up to 80 percent faster for sliding windows. This obviates the need for non-incremental streaming window aggregation and, thus, holds the potential to not only make stream processing systems faster but simpler as well. Our algorithms can be integrated into JVM-based stream processing systems, such as Apache Flink [6], Spark Streaming [27] or SABER [14], by utilising the Java Native Interface (JNI) calls and the Java NIO Direct Buffers.

We consider the presented study the first step towards a highly CPU efficient stream processing system. Naturally, many components of that system are still missing. For example, we will develop a complete set of highly efficient streaming operators on a variety of computer architectures. Already previous work considered porting sliding window aggregate operators on an FPGA [17] or a GPU [14]. For performance portability, however, we believe that the *just-in-time* generation of platform-specific, executable code is essential. To that end, we envision a set of hardware-oblivious

primitives in line with recent work in the field of relational data processing [19]. This way, a compiler can generate query implementations that match the performance of the presented, pre-generated implementations. Moreover, we want to take into consideration the non-uniform memory access (NUMA) [28] introduced by multiple sockets on modern scale-up architectures for our system's design.

7. REFERENCES

[1] Re: Implement a queue in which **push_rear**(), **pop_front**() and **get_min**() are all constant time operations. `http://stackoverflow.com/questions/4802038`, 2018. Online accessed: 27.03.2018.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Is, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[4] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. *Technical Report*, (2004-15):336–347, 2004.

[5] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental sliding window analytics. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 61–72, New York, NY, USA, 2014. ACM.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink : Stream and Batch Processing in a Single Engine. 2015.

[7] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.*, 8(4):401–412, Dec. 2014.

[8] feedzai.com. Modern Payment Fraud Prevention at Big Data Scale, 2013. `http://tinyurl.com/nwnzdxs`.

[9] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In J. Fürnkranz and T. Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning*, ICML'10, pages 13–20, Haifa, Israel, 2010. Omnipress.

[10] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, DaMoN '07, pages 4:1–4:6, New York, NY, USA, 2007. ACM.

[11] M. Hirzel, S. Schneider, and K. Tangwongsan. Sliding-Window Aggregation Algorithms. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17*, pages 11–14, 2017.

[12] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D. (253665), 2016.

[13] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGrid '10, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society.

[14] A. Koliousis, M. Weidlich, R. C. Fernandez, A. Wolf, P. Costa, and P. Pietzuch. Saber: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, 2016. ACM.

[15] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. *Proceedings of the VLDB Endowment*, 8(10):1058–1069, 2015.

[16] 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. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[17] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[18] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[19] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14):1707–1718, 2016.

[20] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[21] Z. Shao. Real-time Analytics at Facebook. Presented at the *Fifth Conference on Extremely Large Databases*, XLDB5, 2011. `http://tinyurl.com/pfcns8q`.

[22] A. Shein, P. Chrysanthis, and A. Labrinidis. FlatFIT: Accelerated incremental sliding-window aggregation for real-time analytics. *ACM International Conference Proceeding Series*, Part F1286, 2017.

[23] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. (Section 4):397–408, 2018.

[24] K. Tangwongsan and M. Hirzel. General Incremental Sliding-Window Aggregation. *Pvldb*, 8(7):702–713, 2015.

[25] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17*, pages 66–77, 2017.

[26] J. Wilkes. More Google Cluster Data. Google Research Blog, `http://bit.ly/1A38mfR`, Nov. 2011. Last access: July 20, 2018.

[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013.

[28] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 659–670, April 2017.