LightSaber: Efficient Window Aggregation on Multi-core Processors

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

Alexandros Koliousis*
Graphcore Research
alexandrosk@graphcore.ai

Peter Pietzuch
Holger Pirk
prp@imperial.ac.uk
pirk@imperial.ac.uk
Imperial College London

Abstract

Window aggregation queries are a core part of streaming applications. To support window aggregation efficiently, stream processing engines face a trade-off between exploiting *parallelism* (at the instruction/multi-core levels) and *incremental* computation (across overlapping windows and queries). Existing engines implement ad-hoc aggregation and parallelization strategies. As a result, they only achieve high performance for specific queries depending on the window definition and the type of aggregation function.

We describe a general model for the design space of window aggregation strategies. Based on this, we introduce *LightSaber*, a new stream processing engine that balances parallelism and incremental processing when executing window aggregation queries on multi-core CPUs. Its design generalizes existing approaches: (i) for *parallel* processing, LightSaber constructs a *parallel aggregation tree* (PAT) that exploits the parallelism of modern processors. The PAT divides window aggregation into intermediate steps that enable the efficient use of both instruction-level (i.e., SIMD) and task-level (i.e., multi-core) parallelism; and (ii) to generate efficient *incremental* code from the PAT, LightSaber uses a *generalized aggregation graph* (GAG), which encodes the low-level data dependencies required to produce aggregates over the stream. A GAG thus generalizes state-of-the-art approaches for incremental window aggregation and supports work-sharing between overlapping windows. LightSaber achieves up to an order of magnitude higher throughput compared to existing systems—on a 16-core server, it processes 470 million records/s with 132 μs average latency.

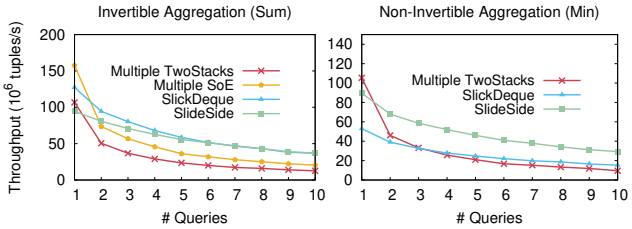


Figure 1: Evaluating window aggregation queries

CCS Concepts

• **Information systems → Data streams**.

Keywords

stream processing; window aggregation; incremental computation

ACM Reference Format:
Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *Proceedings of the 2020 ACM SIGMOD International Conferenceon Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3389753

1 Introduction

As an ever-growing amount of data is acquired by smart home sensors, industrial appliances, and scientific facilities, stream processing engines [5, 10, 40, 68] have become an essential part of any data processing stack. With big data volumes, processing throughput is a key performance metric and recent stream processing engines therefore try to exploit the multi-core parallelism of modern CPUs [50, 76].

In many domains, streaming queries perform *window aggregation* over conceptually infinite data streams. In such queries, a *sliding* or *tumbling* window moves over a data stream while an aggregation function generates an output stream of window results. Given the importance of this class

*Work done while at Imperial College London

SIGMOD'20, June14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389753

of query, modern stream processing engines must be designed specifically for the efficient execution of many window aggregation queries on multi-core CPUs.

Window aggregation queries with tumbling windows process data streams in non-overlapping batches, which makes them amenable to the same types of efficient execution techniques as classic relational queries [24, 57, 60]. In contrast, sliding windows offer a new design challenge, which has not been explored comprehensively. When executing a query with sliding window aggregation, we observe a tension between techniques that use (i) *task-* and *instruction-level parallelism*, leveraging multiple CPU cores and SIMD instructions; and (ii) *incremental processing*, avoiding redundant computation across overlapping windows and queries. Incremental processing introduces control and data dependencies among CPU instructions, which reduce the opportunities for exploiting parallelism.

Current designs for stream processing engines [5, 10, 40, 68] pick a point in this trade-off space when evaluating window aggregation queries. Consequently, they do not exhibit robust performance across query types. Fig. 1 illustrates this problem by comparing state-of-the-art approaches (described in §2.1 and §7.7) for the evaluation of window aggregation queries [2, 29, 62, 64, 67]. The queries, taken from a sensor monitoring workload [34], calculate a rolling sum (invertible) and min (non-invertible) aggregation, with uniformly random window sizes of [1, 128K] tuples and a worst-case slide of 1. Some approaches exploit the invertibility property [29, 62] to increase performance; others [64, 67] efficiently share aggregates for non-invertible functions. To assess the impact of overlap between windows, we increase the number of concurrently executed queries.

As the figure shows, each of the four approaches outperforms the others for some part of the workload but is suboptimal in others: on a single query, the SoE and TwoStacks algorithms perform best for invertible and non-invertible functions, respectively; with multiple overlapping queries, SlickDeque and SlideSide achieve the highest throughput for invertible functions, while SlideSide is best in the non-invertible case. We conclude that *a modern stream processing engine should provide a general evaluation technique for window aggregation queries that always achieves the best performance irrespective of the query details.*

The recent trend to implement query engines as code generators [52, 59] only amplifies this problem—with the elimination of interpretation overhead, differences in the window evaluation approaches have a more pronounced effect on performance. Generating executable code from a relational query is non-trivial (as indicated by the many papers on the matter [52, 55, 59]), but fundamentally a solved problem: most approaches implement a variant of the compilation algorithm by Neumann [52]. No such algorithm, however, exists when overlapping windows are aggregated incrementally. This is challenging because code generation must be expressive enough to generalize different state-of-the-art approaches, as used above.

A common approach employed by compiler designers in such situations is to introduce an abstraction called a "stage"—an intermediate representation that captures all of the cases that need to be generated beneath a unified interface [55, 58]. The goal of our paper is to develop just such a new abstraction for the evaluation of window aggregation queries. We do so by dividing the problem into two parts: (i) effective parallelization of the window computation, and (ii) efficient incremental execution as part of code generation. We develop abstractions for both and demonstrate how to combine them to design a new stream processing system, LightSaber. Specifically, our contributions are as follows:

(i) Model of window aggregation strategies. We formalize window aggregation strategies as part of a general model that allows us to express approaches found in existing systems as well as define entirely new ones. Our model splits window aggregation into intermediate steps, allowing us to reason about different aggregation strategies and their properties. Based on these steps, we determine execution approaches that exploit instruction-level (i.e., SIMD) as well as task-level (i.e., multi-core) parallelism while retaining a degree of incremental processing.

(ii) Multi-level parallel window aggregation. For the parallel computation of window aggregates, we use a *parallel aggregation tree* (PAT) with multiple query- and data-dependent levels, each with its own parallelism degree. A node in the PAT is an execution task that performs an intermediate aggregation step: at the lowest level, the PAT aggregates individual tuples into partial results, called panes. Panes are subsequently consumed in the second level to produce sequences of window fragment results, which are finally combined into a stream of window results.

(iii) Code generation for incremental aggregation. To generate executable code from nodes in the PAT, we propose a *generalized aggregation graph* (GAG) that exploits incremental computation. A GAG is composed of nodes that maintain the aggregate value of the data stored in their child nodes. It can, thus, efficiently share aggregates, even with multiple concurrent queries. By capturing the low-level data dependencies of different aggregate functions and window types, the GAG presents a single interface to the code generator. The code generator traverses an initially unoptimized GAG and specializes it to a specific aggregation strategy by removing unnecessary nodes.

Based on the above abstractions, we design and implement LightSaber, a stream processing system that balances parallelism and incremental processing on multi-core CPUs. Our

Algorithm	Queries	Time				Space	
		single		multiple		single	multiple
		amort.	worst	amort.	worst		
SoE [29]	inv.	2	2	q	q	n	qn
	non-inv.	n	n	qn	qn	n	qn
TwoStacks [29]		3	n	q	qn	$2n$	$2qn$
Slick-Deque [62]	inv.	2	2	$2q$	$2q$	n	$q+n$
	non-inv.	<2	n	q	qn	2 to $2n$	2 to $2n$
Slide-Side [67]	inv.	3	n	q	q	3n	$3n$
	non-inv.	3	n	q	qn	$2n$	$2n$
FlatFAT [63]		$\log(n)$	$\log(n)$	$q\log(n)$	$q\log(n)$	$2n$	$2n$

Table 1: Complexity of window aggregation approaches (n partial aggregates, q queries)

experimental evaluation demonstrates the benefits of PATs and GAGs: LightSaber outperforms state-of-the-art systems by a factor of seven for standardized benchmarks, such as the Yahoo Streaming Benchmark [23], and up to one order of magnitude for other queries. Through micro-benchmarks, we confirm that GAGs generate code that achieves high throughput with low latency compared to existing incremental approaches. On a 16-core server, LightSaber processes tuples at 58 GB/s (470 million records/s) with 132 μs latency.

The paper is organized as follows: §2 explains the problem of sliding window aggregation, the state-of-the-art in incremental window processing, and our model for window aggregation; after that, we describe LightSaber's parallel aggregation tree (§3) and code generation approach (§4) based on a generalized aggregation graphs (§5), followed by implementation details (§6); we finish with the evaluation (§7), related work (§8), and conclusions (§9).

2 Background

In this section, we cover the underlying concepts of window aggregation required for the remainder of the paper. We provide background on window aggregation (§2.1) and its implementation in current systems (§2.2). We finish with a model of the design space for window aggregation (§2.3).

2.1 Window aggregation

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*. We focus on two window types [4]: *tumbling* windows divide the input stream into segments of a fixed-size length (i.e., a static *window size*), and each input tuple maps to exactly one window instance; and *sliding* windows generalize tumbling windows by specifying a *slide*. The slide determines the distance between the start of two windows and allows tuples to map to multiple window instances. Windows are considered *deterministic* [16] if, when a tuple arrives, it is possible to designate the beginning or end of a window.

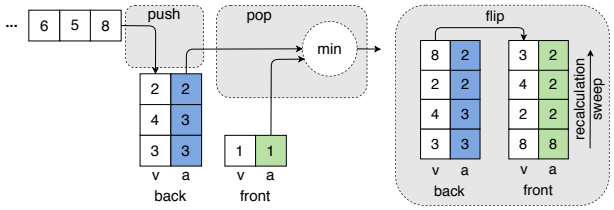


Figure 2: TwoStacks algorithm

An *aggregation function*, such as sum or max, performs arbitrary computation over the window contents to generate a window aggregate. Aggregation functions can be classified based on their algebraic properties [27, 63]: (i) associativity, $(x \oplus y) \oplus z = x \oplus (y \oplus z), \forall\, x, y, z$; (ii) commutativity, $x \oplus y = y \oplus x, \forall\, x, y$; and (iii) invertibility, $(x \oplus y) \ominus y = x, \forall\, x, y$.

Partial aggregation. A typical execution strategy for window aggregation exploits the associativity of aggregation functions: tuples can be (i) partitioned logically, (ii) pre-aggregated in parallel, and (iii) the per-partition aggregates can be merged. This technique is known as hierarchical/partial aggregation in relational databases [12, 25] and *window slicing* in streaming. A number of different slicing techniques have been proposed, e.g., Panes [45], Pairs [41], Cutty [16], and Scotty [69], which remove redundant computation steps by reusing partial aggregates. To further improve performance with overlapping windows, slices can be pre-aggregated incrementally to produce higher-level aggregates. We refer to these window fragment results as *sashes*.

Incremental aggregation. Although windows are finite subsets of tuples, their size can be arbitrarily large. It, therefore, is preferable to use partial aggregation combined with incremental processing to reduce the number of operations required for window evaluation. Depending on the aggregation function, different algorithms can be used to aggregate elements or partial aggregates incrementally.

Table 1 provides an overview of different incremental aggregation algorithms: Subtract-on-Evict (SoE) [29] reuses the previous window result to compute the next one by evicting expired tuples and merging in new additions. This has a constant cost per element for invertible functions but rescans the window if the functions are non-invertible.

For non-invertible functions, TwoStacks [2, 29] achieves O(1) amortized complexity. As shown in Fig. 2, it maintains a *back* and a *front* stack, operating as a queue, to store the input values (white column) and the aggregates (blue/green columns). Each new input value v is *pushed* onto the back stack, and its aggregate is computed based on the value of the back stack's top element. When a *pop* operation is performed, the top of the front stack is removed and aggregated with the top of the back stack. When the front stack is empty, the algorithm *flips* the back onto the front, reversing the order of the values, and recalculates the aggregates.

System	Shared memory	Parallelization	Incremental	Slicing/query sharing
Flink [5], *Spark* [10]	✗	partition-by-key	within window	✗
Cutty [16], *Scotty* [69]	✗	partition-by-key	within/across window	✓
StreamBox [50], *BriskStream* [76]	✓	partition-by-key	within window	✗
SABER [39]	✓	late merging [43, 75], single-threaded merge	within/across window	✗
LightSaber	✓	late merging [43, 75], **parallel merge**	within/across window	✓

Table 2: Window aggregation in stream processing systems

While the previous algorithms process only a single query, another set of algorithms shares the work between multiple queries on the same stream. For multiple invertible functions, SlickDeque generalizes SoE by creating multiple instances of SoE that share the same input values; for non-invertible functions, instead of using two stacks to implement a queue, SlickDeque uses a deque structure to support insertions/removals of aggregates. It, therefore, reports results in constant amortized time. FlatFAT stores aggregates in a pointer-less binary tree structure. It has $O(\log n)$ complexity for updating and retrieving the result of a single query by using a *prefix*- [14] and *suffix*-scan over the input. SlideSide uses a similar idea, but computes a running *prefix/suffix*-scan with O(1) amortized complexity.

As our analysis shows, while the most efficient algorithms have a O(1) complexity, they only achieve this for specific window definitions. We conclude that there is a need to generalize window aggregation across query types.

2.2 Streaming processing engines

By its very nature, window aggregation can be executed either in parallel, if there is independent work, or incrementally, which introduces dependent work—but not both. Consequently, the benefits of work-sharing between overlapping windows must be traded off against the benefits of parallelization. Table 2 summarizes the design decisions of existing stream processing systems. The second column denotes whether a system considers shared-memory architectures; the "parallelization" column specifies how computation is parallelized; the "incremental" column describes when incremental computation is performed; and the last column considers slicing/inter-query sharing.

Scale-out systems (Spark [10] and Flink [5]) distribute processing to a shared-nothing cluster and parallelize it with key-partitioning. This requires a physical partitioning step between each two operators that enables both intra- and inter-node parallelism. However, with this approach, not only significant partitioning overhead is introduced, but also the parallelism degree is limited to the number of distinct aggregation keys, which reduces performance for skewed data. In terms of incremental computation, these systems aggregate directly individual tuples into full windows, following the bucket-per-window approach [46, 47]. This becomes expensive for sliding windows with a small slide when a single input tuple contributes to multiple windows.

Slicing frameworks, like Cutty [16] and Scotty [69] developed on top of Flink, reduce the aggregation steps for sliding windows and enable efficient inter-query sharing. Eager slicing [16] performs incremental computation both within and across window instances using FlatFAT; lazy slicing [69] evaluates only the partial aggregates incrementally and yields better throughput at the cost of higher latency.

Scale-up systems (StreamBox [50] and BriskStream [76]) are designed for NUMA-aware processing on multi-core machines. Both systems parallelize with key-partitioning and use the bucket-per-window approach for incremental computation, overlooking the parallelization and incremental opportunities of window aggregation.

SABER [39] is a scale-up system that parallelizes stream processing on heterogeneous hardware. Instead of partitioning by key, it assigns micro-batches to worker threads in a round-robin fashion, which are processed in parallel but merged in-order by a single thread. This decouples the window definition from the execution strategy and, thus, permits even windows with small slides to be supported with full data-parallelism, in contrast to slice-based processing [11, 45]. SABER decomposes the operator functions into: a *fragment function*, which processes a sequence of window fragments and produces fragment results (or *sashes*); and an *assembly function* (late merging [43, 75]), which constructs complete window results from the fragments. In terms of incremental computation, SABER shares intermediate results both within and across window instances at a tuple level, which results in redundant operations.

While most of these systems apply key-partitioning for parallelization, this approach does not exploit all the available parallel hardware. In addition, when evaluating overlapping windows, no system from Table 2 combines effectively partial aggregation with incremental computation, which results in sub-optimal performance. It is crucial, hence, to design a system based on these observations.

2.3 Window aggregation model

As described earlier, the properties of the aggregation functions can be exploited to compute aggregates either in parallel or incrementally. Surprisingly, we found that current stream processing systems do not take advantage of this. Evidently, there is a design space for stream processing systems without a dominating strategy for window aggregation. What is lacking, therefore, is a model that captures the design space and allows reasoning about design decisions and their impact on system performance.

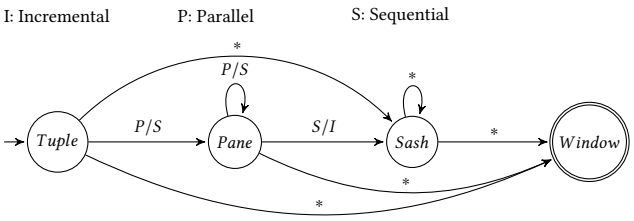


Figure 3: Model of window aggregation design space

In this model, an aggregation strategy is represented as a word over the alphabet of intermediate result classes, *Pane*, *Sash*, *Window*, and combiner strategies, *parallel* (*P*), *incremental* (*I*), and *sequential* (*S*). The DFA in Fig. 3 shows the possible sequences of intermediate steps to produce a window aggregate from a set of tuples. From left to right, tuples are aggregated into *Panes* (or slices [16]), which can be merged and aggregated into *Sashes* in one or more (i.e., hierarchical) steps. *Sashes* are combined into complete *Windows* (also, potentially hierarchically). Conceptually, each of those aggregation steps can be *I*, *P*, or *S*.

Given this model, a system is described by a word of the form ($S \rightarrow Pane, PPI \rightarrow Sash, I \rightarrow Window$).[1] The previous word encodes a design in which tuples are aggregated into *Panes* sequentially; *Panes* are hierarchically aggregated into *Sashes* in two parallel and one incremental step; and the final windows are produced in a single incremental step.

To illustrate this further, let us encode a number of real systems in the model. Systems that utilize the bucket-per-window approach, such as Flink, have an aggregation tree that has only one (incremental) level ($I \rightarrow Window$) in our model. Slicing [16, 69] removes redundant computations and is either ($S \rightarrow Pane, I \rightarrow Window$) or ($S \rightarrow Pane, S \rightarrow Window$). SABER is a specialized stream processing system that has three processing states: it incrementally aggregates tuples into *Sashes*, aggregates multiple *Sashes* in parallel but generates complete *Windows* sequentially. Formally, SABER implements ($I \rightarrow Sash, PS \rightarrow Window$).

SABER, arguably the most advanced of these systems, still does not exploit all available parallelism: (i) it does not parallelize slice creation and (ii) because the aggregation strategy is fixed, the degree of parallelism is determined by the window definition, which limits scalability. In the next section, we describe how LightSaber overcomes these limitations by occupying a new point in this design space.

3 Parallel Aggregation Tree

Having formalized the design space, let us, now, describe how to select an appropriate design. The first thing to note is

[1]Note that spaces, commas, and → are merely for readability and do not have formal semantics.

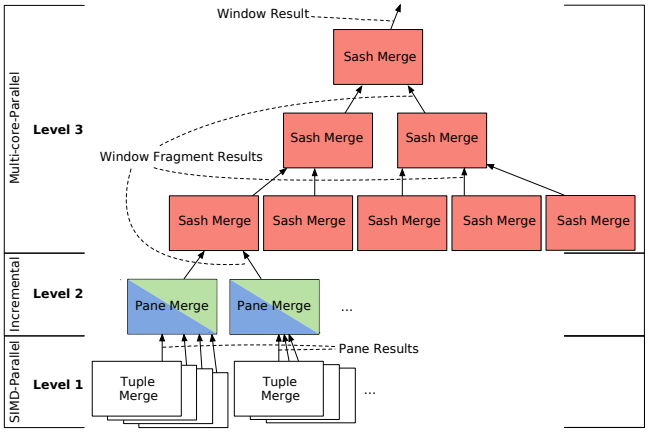


Figure 4: Parallel aggregation tree in LightSaber

that the optimal design is highly query-, input- and system-specific. In particular, the aggregation from sashes to windows can be performed hierarchically (i.e., following the self-referencing loop at the sash-node in Fig. 3 multiple times). The optimal number of hierarchical merge steps is difficult to determine statically. For this reason, we determine the number of hierarchical merge steps (i.e., based on the runtime state of the system). To express this fact, we use the Kleene-star as a suffix to a literal encoding the processing strategy: I^*, e.g., indicates a dynamically determined number of incremental aggregations. In this formalism, LightSaber is a ($P \rightarrow Pane, I \rightarrow Sash, P^*S \rightarrow Window$) system: parallel aggregation of tuples into panes, incremental aggregation of panes into sashes and a dynamic number of parallel aggregations with a final sequential step to produce windows from sashes.

Implementing such a design, however, is non-trivial because it combines static subplans (the tuples to panes and panes to sashes) with dynamic ones (sashes to windows). For that reason, the plan is encoded in a structure we call a *Parallel Aggregation Tree* (PAT) (see Fig. 4). A PAT has three distinct levels, one for each intermediate result class in the formalism. While the two bottom levels are statically defined based on the properties of the query, the depth of the last level is data-dependent and can have arbitrarily many layers, as $P*$ indicates. This increases the degree of parallelism available in the workload, which allows LightSaber to scale to more parallel hardware for queries that do not come with high degrees of inherent parallelism (i.e., those with either an expensive aggregate combiner function or a small slide). Next, we describe each of the processing levels of the PAT in more detail.

Level 1: Tuple merge. The goal of the first level is to aggregate the tuples of a pane. Since there is no sharing potential, this level can be parallelized without any downsides. LightSaber partitions the stream into panes similar to the strategy pioneered by Pairs [41] or Cutty [16]. Within each

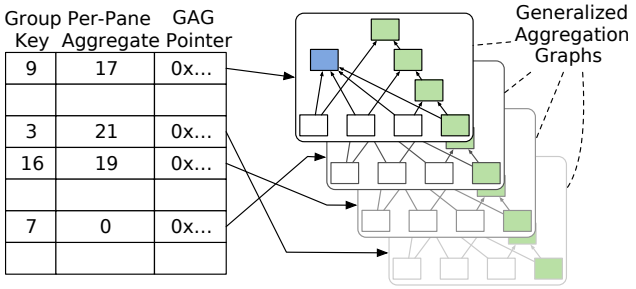


Figure 5: Pane aggregation table

pane, LightSaber exploits SIMD as well as instruction-level parallelism in line with previous work [66].

The panes are computed using a specialized data structure, which we call a *pane aggregation table* (see Fig. 5). The *pane aggregation table* is shared between the two bottom levels of Fig. 4. For each active key, it maintains a separate pane result and a *generalized aggregation graph* (GAG) instance. GAGs are an abstract data structure supporting incremental aggregation of values and will be described in §5. GAGs are abstract in the sense that their implementation is generated at query compilation time. The result of the tuple level merge is either a hashtable with per-key aggregates or a single aggregate value for ungrouped aggregations.

Level 2: Pane merge. The pane merge level combines the window panes into sashes. It does so by iterating over the pane aggregation table and merging each per-pane aggregate into a GAG. With an interface similar to a queue, GAGs support the insertion of a value, which triggers the emission of a window aggregate. If a pane produces no results for a key, the neutral element for the combiner function is merged into the GAG to trigger the emission of an aggregate. To reduce the memory pressure, LightSaber maintains a timeout value for each key, which marks the time at which it no longer contributes to windows and is safe to remove it.

While the evaluation of the tuple merge level can be an individual task and the merging of values into the GAG can be parallelized across CPU cores, LightSaber combines the bottom levels of the PAT and generates code that is executed as a single, fully-inlined task at runtime. This reduces the number of function calls and improves data and instruction locality. SIMD parallelization, however, is an interesting optimization, which we leave for future work. The output of this level is a sash, which is passed to the next level using a simple result buffer.

Level 3: Sash merge. In the final level of the PAT, the sash results are combined and emitted as a stream of complete window results. At first, each window fragment is tagged as opening, closing, complete, or pending, similar to SABER [39]. Instead of performing a single-threaded merge though, LightSaber merges the window fragments in an aggregation tree: for each pair of sashes, a task is created with the respective aggregate combiner function and a pointer to the sashes. Each task designates one of its inputs as the "higher-level aggregate" side and merges the values from the other input into it to decrease the memory footprint of this level. LightSaber keeps track of the active window fragments' dependencies. When the merge is done, it returns the intermediate buffers to statically allocated pools of objects. The worker that merges the last two windows emits its output as the complete window result.

4 Query Code Generation

LightSaber generates code for two separate purposes: at the query level, code is generated to implement the semantics (filters, joins, etc.) of the query but abstracts away the incremental aggregation strategy (SoE, TwoStacks, etc.); at the level of aggregation strategies, the query semantics are abstracted. The two levels are connected by the pane aggregation table, as discussed in the previous section. We first discuss the query-level code generation in this section and the incremental aggregation in §5.

4.1 Operators

Let us define the set of operators supported by LightSaber: (i) the `PROJECTION` (π) and `SELECTION` (σ) operators, which are stateless and require a single scan over the stream batch; (ii) the `WINDOW AGGREGATION` (α) operator for both tumbling and sliding windows, which supports `GROUP-BY` (γ) as well as standard aggregation functions (`min`, `max`, `sum`, `count`, `average`); and (iii) the `JOIN` ($\bowtie$) operator, which allows joining a stream with a static table.

`GROUP-BY` and `JOIN` are implemented using generated (and, thus, inlined) hashtables as well as aggregation code. Following common practice, we use Intel's SSE CRC instruction [38] and bit masking for hashing, open addressing for hashtable layout, and linear probing for conflict resolution. Hashtables are pre-allocated according to (generous) cardinality estimates but can be resized (and rehashed on overflow).

4.2 Generating query code

To translate the non-incremental operators to executable code, we follow the approach by HyPeR [52]: we traverse the operator tree and append instructions to a code buffer until reaching an operator that requires materialization of its result (a pipeline breaker). Since LightSaber does not currently support stream/stream joins, the only pipeline breakers are windowed aggregations. Every sequence of pipelineable operators translates to a single worker task, which minimizes task overhead (i.e., allocation, result passing, and scheduling). Each task is optimized and compiled into executable code by the LLVM compiler framework [42].

To illustrate this process, consider the query in Fig. 6a (denoted in Continuous Query Language Syntax [7]). Taken

```
select timestamp, highway, direction,
    (position /5280) as segment, AVG (speed) as avgSpeed
from PosSpeedStr [range 300 slide 1]
group by highway, direction, segment
having avgSpeed < 40.0
```

(a) SQL for LRB_1 query

```
1  paneAggregationTable.reset_panes();
2  for (tuple &t: input) {
3    if (paneAggregationTable.pane_has_ended(t)) {
4      sashes.append(paneAggregationTable.extract_sashes());
5      paneAggregationTable.reset_panes();
6    }
7    key = {t._1, t._3, t._5, (t._6 / 5280)};
8    val = t._2;
9    paneAggregationTable.tuple_merge(key, val);
10 }
```

(b) Generated code for fragment function of LRB_1

```
1 size_t pos = find(key);
2 hashNode[pos].pane._1 += val;
3 hashNode[pos].pane._cnt++;
```

(c) Function tuple_merge(key, val)

```
1 sashesTable;
2 for (/* iterate i over the hash nodes*/) {
3   gag[i].evict();
4   val = {hashNode[i].pane_1, hashNode[i].pane._cnt};
5   gag[i].insert(val);
6   sashesTable.append(gag[i].query(WINDOW_SIZE));
7 }
8 return sashesTable;
```

(d) Function extract_sashes()

Figure 6: Query code generation in LightSaber

from the Linear Road Benchmark [8], it reports the road segments on a highway lane with an average speed lower than 40 over a sliding window. A simplified version of the generated code for the generated *fragment function* of this query (in C++ notation) is shown in Fig. 6b. Note that the LightSaber query compiler combines levels 1 and 2, as illustrated in Fig. 4, into a single, fully-inlined task (line 9 implements level 1, lines 4-5 implement level 2). The generated query code reflects only the query semantics (projection, grouping key calculation) but expresses aggregation purely in terms of the pane aggregation table API. The Pane Aggregation Table implementation (Figures 6c and 6d) is a very thin wrapper over the GAGs: the `tuple_merge` function acts like an ordinary hashtable while the `extract_sashes` function spills the pre-aggregated pane results into the GAG using three functions: `insert`, `evict` and `query`. Let us, in the next section, discuss the GAG interface as well as its implementation.

5 Generalized Aggregation Graph

The objective of GAGs is to combine the benefits of code generation (hardware-conscious, function-call-free code) with those of efficient incremental processing. While this is achievable by hard-coding query fragments in the form of "templates" and instantiating them at runtime, this approach quickly leads to unmaintainable large codebases and complicated translation rules. Instead, a code generation approach should be based on an abstraction that is expressive enough

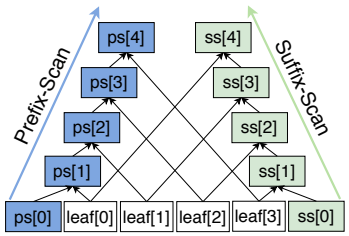


Figure 7: Initial General Aggregation Graph

to capture all of the targeted design space yet simple enough to maintain. The focus of this section is the development of an abstraction that captures the design space of the best (known) in-order incremental processing algorithms.

Before presenting the intuition behind the GAG representation, it is necessary to provide the definitions of the *prefix-* [14] and the *suffix-scan*. Given an associative operator $\oplus$ with an identity element $I_\oplus$, and an array of elements $A[a_0, a_1, ..., a_{n-1}]$, the *prefix-* and *suffix-scan* operations are defined as $PS[I_\oplus, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$ and $SS[I_\oplus, a_{n-1}, (a_{n-1} \oplus a_{n-2}), ..., (a_{n-1} \oplus a_{n-2} \oplus ... \oplus a_0)]$ respectively. The first $n-1$ elements of these arrays represent an exclusive `prefix-` or `suffix-scan`, while the elements without the identity $I_\oplus$ represent an inclusive scan.

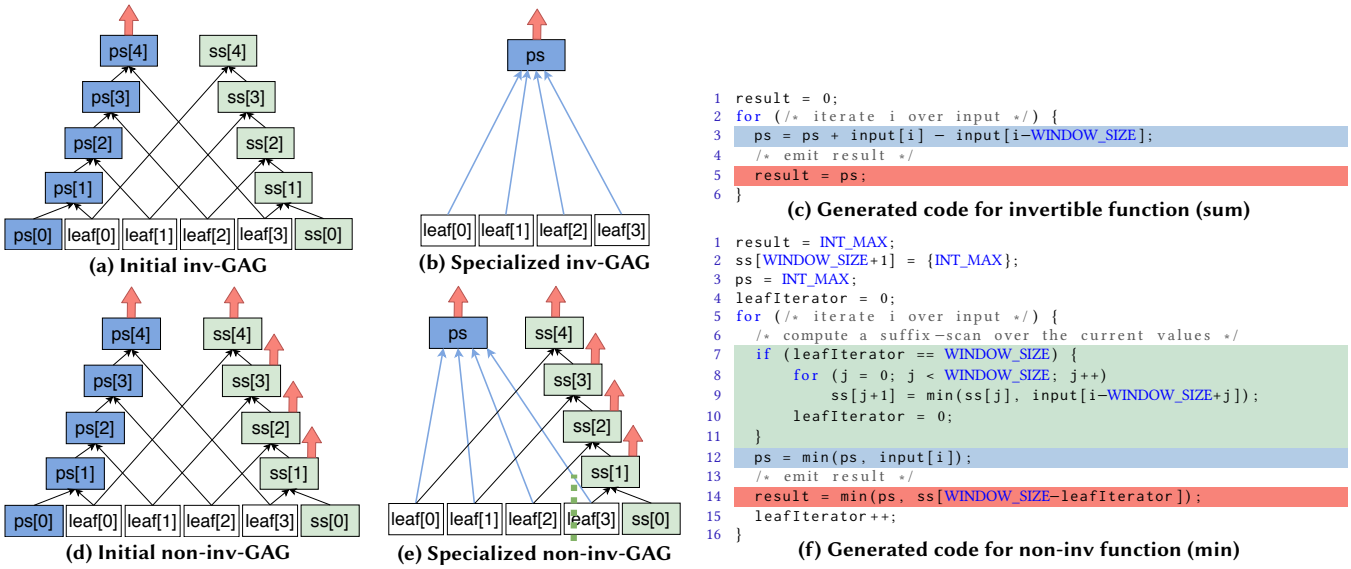
Let us, now, discuss the different aspects of GAGs in the order that they become relevant in the code generation process: starting with the interface, the creation of an initial generic GAG, the specialization to a specific aggregation strategy, and the translation into executable code. We also discuss an optimization for multi-query processing as well as the applicability of the GAG approach.

GAG interface As discussed in the previous section, the generated code of GAGs relies on three functions to enable efficient shared aggregation:[2]

- `void insert (Value v)`: inserts an item of type `Value` in the GAG and performs any internal changes necessary to accommodate further operations;
- `void evict ()`: removes the oldest value and perform the necessary internal changes; and
- `Value query (size_t windowSize)`: returns the result with respect to the current state and a given window size.

While the first two are rather obvious, the `query` function is interesting in that it indicates that GAG produces results for different window sizes. Such inter-query sharing requires to maintain partial aggregates in memory and support in-order range queries efficiently. LightSaber generates shared partials (see Fig. 6b) and GAGs take care of storing them and producing window results by invoking the `query` function.

[2]Note that these functions are conceptual and do not exist in generated code.




```
result = 0;
for (/* iterate i over input */) {
  ps = ps + input[i] - input[i-WINDOW_SIZE];
  /* emit result */
  result = ps;
}
```

(c) Generated code for invertible function (sum)

```
result = INT_MAX;
ss[WINDOW_SIZE+1] = {INT_MAX};
ps = INT_MAX;
leafIterator = 0;
for (/* iterate i over input */) {
  /* compute a suffix-scan over the current values */
  if (leafIterator == WINDOW_SIZE) {
      for (j = 0; j < WINDOW_SIZE; j++)
          ss[j+1] = min(ss[j], input[i-WINDOW_SIZE+j]);
      leafIterator = 0;
  }
  ps = min(ps, input[i]);
  /* emit result */
  result = min(ps, ss[WINDOW_SIZE-leafIterator]);
  leafIterator++;
}
```

(f) Generated code for non-inv function (min)

Figure 8: Generating code with GAGs

5.1 Initial GAG creation

A GAG must turn a window definition (size, slide, and aggregation function) into a dataflow graph that can be used to produce any of the presented algorithms in Table 1: SoE, TwoStacks, SlickDeque, and SlideSide (since each of these is the best-performing in parts of the problem space). At first glance, this seems to be a challenging problem that requires a complex solution. There is, however, an observation that turns this problem into an almost trivial one: any associative aggregation function can be represented as a binary combination of an element of a *prefix-scan* and an element of a *suffix-scan* of the input. This is sufficient to capture the low-level dependencies of the aforementioned algorithms.

Tangwongsan et al. [63] make a similar observation when developing FlatFAT, a binary aggregation tree (see §2.1). While FlatFAT performs poorly due to the runtime overhead of the tree, we found that the principle can be applied to generalize a data structure that is efficient for non-invertible combiners: TwoStacks [29]. This results in a dataflow graph as the one in Fig. 7, which can represent any associative aggregation. As indicated by the color-coded nodes (which mirror those of Fig. 2), the front stack corresponds to the blue *prefix-tree* parent nodes (abbreviated as `ps`). The `ps` values effectively constitute a running *prefix-scan* over the input values (`leaves` in the graph, which can be panes or individual elements). On the right-hand side in Fig. 7, (green) *suffix-scan* parents (`ss`) correspond to the back stack.

Extracting an aggregate from this graph is as simple as combining the two nodes from the *prefix-* and *suffix-scan* using the appropriate combiner function. These nodes' values can be updated either lazily (upon query) or eagerly (upon tuple insertion). Next, we describe how we exploit this property to specialize the initial GAG into one of the presented aggregation algorithms.

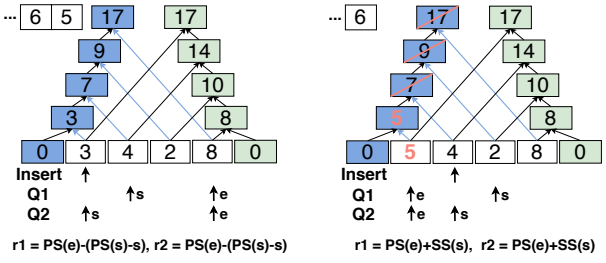
5.2 GAG specialization

The first step in GAG specialization is the removal of unnecessary nodes, which would lead to "dead code". This is a simple "mark and sweep" optimization: every node that is needed to produce final aggregates is marked as required. Afterwards, the GAG is traversed top-down, and each unnecessary node is replaced by its children. If multiple nodes have the same inputs, they are eliminated as duplicates.

The second step is the definition of a lazy or eager dataflow computation strategy. The rationale is simple: prefix-scans can be efficiently calculated without buffering (i.e., eagerly) because they only require access to the last element and the current one; suffix-scans require buffering because they need to access older tuples from the stream. In addition, suffix-scans must scan the window, which incurs a linear cost. It is, therefore, beneficial to perform the suffix-calculation lazily. This process is required whenever evictions are performed. We refer to the respective tuple ingestions as a "trigger-point" and represent it with a dashed vertical line in Fig. 8e. Note that the suffix-scan directly corresponds to the "stack-flipping" of the TwoStacks algorithm.

To illustrate the (somewhat surprising) power of this approach, let use describe the two most illustrative cases: single-query aggregation using an invertible (Fig. 8b) and a non-invertible (Fig. 8e) combiner function. Note that we draw *eager* computation edges in blue and *lazy* ones in black.

Fig. 8a shows *the case of an invertible function*. Here only the root of the *prefix-scan* is marked as required (indicated by the red arrow)—all other values are unnecessary and can



(a) GAG at time 0 (b) GAG at time 1

Figure 9: Example of multi-window processing

be eliminated. After removing all unmarked nodes and replacing the parent edges with their children, only a single node remains (Fig. 8b). This corresponds to the dataflow representation of the SoE algorithm.

In the case of a non-invertible function (Fig. 8d), each of the values of the *suffix-scan*, as well as the root of the *prefix-tree*, are required. Only the *prefix-scan* tree can be collapsed, resulting in the dataflow graph in Fig. 8e. Since the graph has lazy (black) compute-edges, it requires a "trigger-point" (indicated by the dashed green line). This graph corresponds to the `TwoStacks` algorithm.

5.3 Generating code from GAGs

Processing a single sliding window while computing an invertible function is the simplest case. The code (displayed in Fig. 8c) simply iterates over the input and, for each tuple, adds the current value to an accumulator while evicting the appropriate value (line 3). Note that, for clarity, the code accesses the evicted value directly from the input stream. In practice, LightSaber buffers only a window's worth of values. After the value is processed, a result is emitted (line 5).

The case of a single window computing a non-invertible function is more complicated. Conceptually, it corresponds to the TwoStacks algorithm (in particular its implementation in HammerSlide [66]): as illustrated in Fig. 8f, the implementation maintains a single aggregate value for the back stack (line 12) as well as a buffer for the back stack. Whenever the number of inserted elements is equal to the window size, it triggers the computation of the suffix-scan (lines 7 through 11). The result is emitted in line 14.

5.4 Multi-window processing

The naïve approach to evaluate multiple concurrent window queries would be to have several instances of the single-window code. Instead, we find that we can answer multiple different queries (over the same stream) by extracting and combining specific values from the GAG. Unfortunately, determining which values to combine is challenging: the problem stems from the fact that tuples are stored in a circular buffer, which leads to the *start* and *end* cursors of a window change their relative order while processing the stream. This is referred to as the inverted buffer problem [63].

To illustrate this problem, consider the example in Fig. 9a: it shows the evaluation of two queries, Q1 and Q2, both calculating a window sum but with different window sizes (3 and 4 elements, respectively). Fig. 9a shows the state at time 0. Q1's window contains the values $\{4, 2, 8\}$. Their sum (i.e., 14) can be calculated by subtracting the exclusive prefix-scan of the window start cursor, $\text{PS}(s) - s$, from the inclusive prefix-scan of the end, $\text{PS}(e)$. Similarly, the result of Q2 at time 0 is calculated as 17. When transitioning to time 1, the value 5 is inserted at the insert cursor, and all cursors are advanced. The end cursors of both windows are now to the left of the start cursors: the windows are inverted. The sum of the window elements can now be calculated as the prefix-sum of the end, $\text{PS}(e)$, and the suffix-sum of the start, $\text{SS}(s)$.

The case of multiple non-invertible functions generalizes the single non-invertible case, using only the root of the `ps` array. Their key difference is that instead of maintaining a single front stack, the algorithm operates on multiple stacks, one for each query. However, these stacks can be overlayed and start at the same memory address. We omit the details due to space limitations.

5.5 Applicability of GAGs

Before concluding this section, let us discuss the conditions under which GAGs are applicable. In terms of the aggregate functions, they have the same requirements as other approaches (see Table 1): the aggregate functions must be only associative and neither invertible nor commutative. These conditions hold for the standard SQL aggregation functions (min, max, sum, count, avg) as well as many statistical properties (most notably standard deviation) but exclude percentiles [29]. New functions can be added similar to other approaches [16, 63]. The collapsing of the GAG to SoE requires an extra `invert` function during the specialization phase. With respect to ordering, GAGs can handle FIFO windows with "in-order" or "slightly out-of-order" events that end up in the same partial aggregate. Handling full out-of-order data is beyond the scope of this work.

6 LightSaber Engine

Next, we describe the LightSaber engine[3] in more detail. It is implemented in 24K lines of C++14 and uses Intel TBB [32] for concurrent queues and page-aligned memory allocation.

6.1 Memory management

As we demonstrate in §7, the performance of LightSaber is mostly restricted by memory accesses and, hence, LightSaber manages its own memory for improved performance.

[3]The source code is available at https://github.com/lsds/LightSaber

The processing of window operators requires the allocation of memory chunks for storing intermediate window fragments. Dynamic memory allocation on the critical path, however, is costly and reduces overall throughput.

We observe that, for fixed-sized windows, the amount of memory required over time is the same. Thus, based on the window definition and the system batch size, LightSaber infers the amount of memory needed and allocates it once at the start. It uses dynamic memory allocation only when more memory is required. Each worker thread is pinned to a CPU core to limit contention, while it initializes and maintains its own pool of data structures and intermediate buffers. For tasks, LightSaber uses statically allocated pools of objects.

6.2 NUMA-aware scheduling

CPU utilization in a multi-socket NUMA system depends on whether tasks are scheduled efficiently to individual worker threads based on data locality. LightSaber employs NUMA-aware scheduling to reduce remote memory accesses.

A lock-free circular buffer per input stream stores incoming tuples. These buffers are spilled between all available sockets. To balance computation, the size of their fragments vary based on the number of worker threads per node. Each query task has a start and end pointer on the circular buffer with read-only access and a window function pointer. It also contains a free pointer, indicating which memory has been processed and can be freed in the result stage. In contrast to systems such as Flink and Spark, LightSaber delays the computation of windows until the query execution stage, thus avoiding a sequential dispatching stage.

For each NUMA node, LightSaber maintains a separate lock-free task queue. When a new task is created, it is placed based on data locality. A worker thread favors tasks from its local queue and, only when there is no more work available, it attempts to steal from other nodes to avoid starvation.

6.3 Operator optimizations

Existing optimization techniques [7, 30, 71] are applicable to LightSaber. Our current implementation includes the following: it (i) reorders operators according to selectivity and moves more selective operators upstream [71] (e.g., for filter-window commutativity [7]); (ii) inlines and applies the `HAVING` clause when a full window result is constructed; and (iii) reduces instructions by removing components related to cross-process and network communication, that introduce conditional branches [76].

7 Evaluation

We evaluate LightSaber to show the benefits of our window aggregation approach. First, we demonstrate that LightSaber achieves higher performance for a range of real-world query benchmarks. Then, we explore the efficiency of its incremental aggregation approach compared to state-of-the-art techniques.

Datasets		Queries			
Name	# Attr.	Name	Windows	Operators	Values
Cluster Monitoring (CM) [22, 36]	12	CM_1 CM_2	$\omega_{60,1}$ $\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{sum}}$ $\pi, \sigma, \gamma, \alpha_{\text{avg}}$	
Smart Grid (SG) [35]	7	SG_1 SG_2	$\omega_{3600,1}$ $\omega_{128,1}$	π, α_{avg} $\pi, \gamma, \alpha_{\text{avg}}$	
Linear Road Benchmark (LRB) [8]	7	LRB_1 LRB_2	$\omega_{300,1}$ $\omega_{30,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$ $\pi, \gamma, \alpha_{\text{count}}$	
Yahoo Streaming (YSB) [23]	7	YSB	$\omega_{10,10}$	$\sigma, \pi, \bowtie_{\text{relation}}$, $\gamma, \alpha_{\text{count}}$	
Sensor Monitoring (SM) [34]	18	SM_f	*various*	α_f	$f \in$ {sum, min}

Table 3: Evaluation datasets and workloads

7.1 Experimental setup and workloads

All experiments are performed on a server with two Intel Xeon E5-2640 v3 2.60 GHz CPUs with a total of 16 physical cores, a 20 MB LLC cache, and 64 GB of memory. We use Ubuntu 18.04 with Linux kernel 4.15.0-50 and compile all code with Clang++ version 9.0.0 using `-O3 -march=native`.

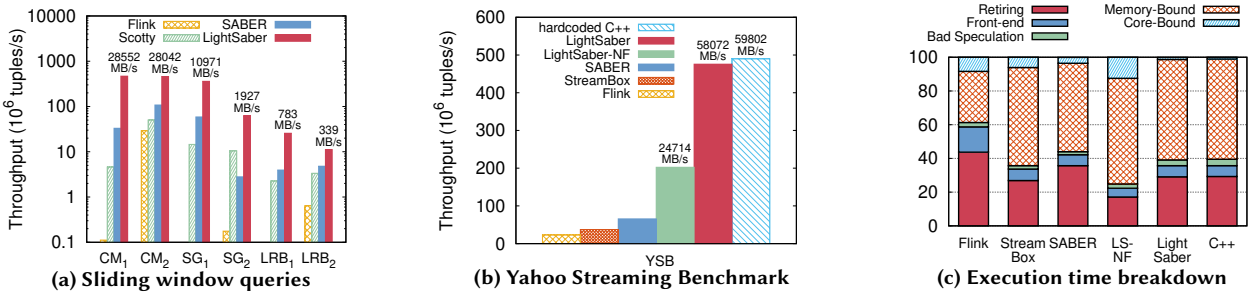
We compare LightSaber against both Java-based scale-out streaming engines, such as Apache Flink (version 1.8.0) [5], and engines for shared-memory multi-cores, such as StreamBox [50] and SABER [39]. For Flink, we disable the fault-tolerance mechanism and enable object reuse for better performance. For SABER, we do not utilize GPUs for a fair comparison without acceleration. To avoid any possible network bottlenecks, we generate ingress streams in-memory by pre-populating buffers and replaying records continuously.

Table 3 summarizes the datasets and the workloads used for our evaluation. The workloads capture a variety of scenarios that are representative of stream processing. In the workloads, window sizes and slides are measured in seconds.

Compute cluster monitoring (CM) [72]. This workload emulates a cluster management scenario using a trace of time-stamped tuples collected from an 11,000-machine compute cluster at Google. Each tuple contains information about metrics related to monitoring events, such as task completion or failure, task priority, and CPU utilization. We execute two queries from previous work [39] to express common monitoring tasks [22, 36].

Anomaly detection in smart grids (SG) [35]. This workload performs anomaly detection in a trace from a smart electricity grid. The trace contains smart meter data from electrical devices in households. We use two queries for detecting outliers: SG_1 computes a sliding global average of the meter load, and SG_2 reports the sliding load average per plug in a household.

Linear Road Benchmark (LRB) [8]. This workload is widely used for the evaluation of stream processing performance [1, 17, 33, 74] and simulates a network of toll roads.



(a) Sliding window queries (b) Yahoo Streaming Benchmark (c) Execution time breakdown

Figure 10: Performance for application benchmark queries

We use queries 3 and 4 from [39], that correspond to LRB_1 and LRB_2 here.

Yahoo Streaming Benchmark (YSB) [23]. This benchmark simulates a real-word advertisement application in which the performance of a windowed count is evaluated over a tumbling window of 10 seconds. We perform the join query, and we use numerical values (128 bits) rather than JSON strings [54].

Sensor monitoring (SM) [34]. The final workload emulates a monitoring scenario with an event trace generated by manufacturing equipment sensors. Each tuple is a monitoring event with three energy readings and 54 binary sensor-state transitions sampled at 100 Hz.

7.2 Window aggregation performance

To study the efficiency of LightSaber in incremental computation, we use six queries from different streaming scenarios, and we compare performance against Flink and SABER. Flink represents the bucket-per-window approach [46, 47] that replicates tuples into multiple window buckets. We use the Scotty [69] approach with Flink to provide a representative system with only the slicing optimization [4]. In contrast, SABER is a representative example of a system that performs incremental computation on per-tuple basis.

Fig. 10a shows that LightSaber significantly outperforms the other systems in all benchmarks. Queries CM_1 and SG_1 have a small number of keys (around 8 for CM_1) or a single key, respectively, which reveals the limitations of systems that parallelize on distinct keys. Flink's throughput, even with slicing, is at least one order of magnitude lower than that of both SABER and LightSaber. This shows that current stream processing systems do not efficiently support this type of computation out-of-the-box, because it requires explicit load balancing between the operators. Compared to SABER, LightSaber achieves 14× and 6× higher throughput for the two queries, respectively, due to its more efficient intermediate result sharing with panes.

For query CM_2, Flink performs better and has comparable throughput to SABER, because of the low selectivity of the selection operator. LightSaber has still 4×, 9× and 15× better performance compared to SABER, Scotty, and Flink, respectively, because it reduces the operations required for window aggregation.

Queries SG_2 and LRB_{1-2} group multiple keys (3 for SG_2 and LRB_1; 4 for LRB_2), increasing the cost of the aggregation phase. In addition, all three queries contain multiple distinct keys, which incurs a higher memory footprint when maintaining the window state. LightSaber achieves two orders of magnitude higher throughput for SG_2 and LRB_1 and 17× higher throughput for LRB_2 compared to Flink, because of the redundant computations. With slicing, Flink has 6×, 11× and 3× worse throughput than LightSaber for the three queries, respectively. Scotty outperforms SABER for SG_2 by 4×, demonstrating how the single-threaded merge becomes the bottleneck. Compared to SABER, LightSaber has 23×, 7× and 2× higher throughput for SG_2, LRB_1, and LRB_2, respectively. This is due to the more efficient partial aggregate sharing, the NUMA-aware placement, and the parallel merge.

7.3 Efficiency of code generation

Next, we explore the efficiency of LightSaber's generated code. Using YSB, we compare LightSaber to Flink, SABER, StreamBox, LightSaber without operator fusion, and a hard-coded C++ implementation. StreamBox is a NUMA-aware in-memory streaming engine with an execution model [43] similar to LightSaber. For this workload, the GAG does not wield performance benefits because there is no potential for intermediate results reuse to exploit. We omit Scotty for the same reason, as slicing does not affect the performance of tumbling windows. Finally, we conduct a micro-architectural analysis to identify bottlenecks.

As Fig. 10b shows, Flink achieves the lowest throughput because of its distributed shared-nothing execution model. A large fraction of its execution time is spent on tuples serialization, which introduces extra function calls and memory copies. For the other systems, we do not observe similar behavior, as tuples are accessed directly from in-memory data structures. LightSaber exhibits nearly 2×, 7×, 12× and 20× higher throughput than LightSaber without operator fusion, SABER, StreamBox, and Flink, respectively. When

[4] Note that we use lazy slicing, which exhibits higher throughput with lower memory consumption.

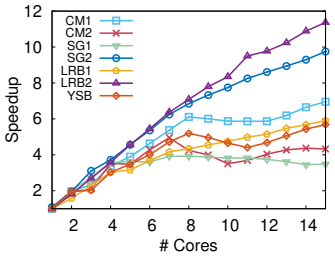


Figure 11: Scalability

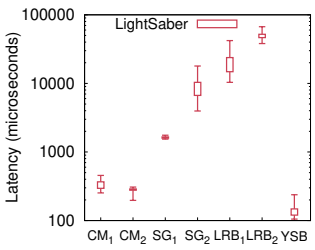
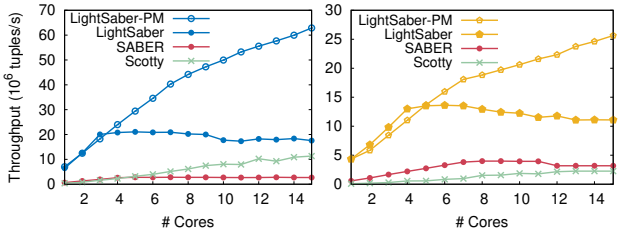


Figure 12: Latency



(a) SG_2 query (b) LRB_1 query

Figure 13: Parallel merge

compared to the hardcoded implementation, we find only a 3% difference in throughput, which reveals the small performance overhead introduced by LightSaber's code generation approach. For the other benchmarks from §7.2, we observe similar results.

Fig. 10c shows a breakdown by CPU components following Intel's optimization guide [31], showing the stalls in the CPU pipeline. The components are categorized as: (i) front-end stalls due to fetch operations; (ii) core-bound stalls due to the execution units; (iii) memory-bound stalls caused by the memory subsystem; (iv) bad speculation due to branch mispredictions; (v) retiring cycles representing the execution of useful instructions.

Flink suffers up to 15% of front-end stalls because of its large instruction footprint. Compared to LightSaber and the hardcoded C++, the other approaches have more core-bound stalls, which indicates that they do not exploit the available CPU resources efficiently. At the same time, all solutions, apart from Flink, are memory-bound but exhibit different performance patterns. Although Streambox is up to 58% memory-bound, its performance is affected by its centralized task scheduling mechanism with locking primitives, and the time spent on passing data between multiple queues. LightSaber without operator fusion exhibits similar behavior and requires extra intermediate buffers that increase memory pressure and hinder scalability. When compared to LightSaber, SABER's Java implementation exploits only 10% of the memory bandwidth, while our system reaches up to 65%. The Java code spends most of the time waiting for data [75] and copying it between operators; LightSaber and the hardcoded C++ implementation utilize all the resources and the memory hierarchy more efficiently. Despite exhibiting better data and instruction locality though, they have the highest bad speculation (up to 4%), because slicing and computation are performed in a single loop.

7.4 Scalability and end-to-end latency

Next, we evaluate the scalability and the end-to-end latency of LightSaber. We use the 7 queries from the previous benchmarks and report the throughput speedup over the performance of a single worker when varying the core count. Note that the first core is dedicated to data ingestion and task creation. We define the end-to-end latency as the time between when an event enters the system and when a window result is produced [70].

The results in Fig. 11 show that LightSaber scales linearly up to 7 cores for all queries, with latencies lower than tens of ms. By conducting a performance analysis of our implementation, we observe that queries CM_{1-2}, SG_1 and YSB do not scale beyond 7 cores, even though the remote memory accesses are kept low. This is the result of the system being memory bound (up to 60%) and operating close to the memory bandwidth. With LightSaber's centralized task scheduling, we observe a throughput close to 400 million tuples/sec and only a 15% performance benefit when we cross NUMA sockets for these four queries. As future work, we want to investigate whether applying our approach to a tuple-at-a-time processing model can yield better results.

On the other hand, for queries LRB_{1-2} and SG_2, we observe up to 3× higher throughput because they are more computationally intensive. In this case, the reduction of the remote memory accesses improves the scalability of LightSaber.

Fig. 12 shows that the average latency remains lower than 50 ms in SG_{1-2} and LRB_{1-2}. The latency is in the order of microseconds for the other queries: for YSB, LightSaber exhibits 132 μs of average latency, which is an order of magnitude lower compared to the reported results of other streaming systems [28, 70]. The main reason for this is that LightSaber combines efficiently partial aggregation with incremental computation, which leads to very low latencies.

7.5 Parallel merging

In queries SG_2 and LRB_1, we group by multiple keys with many distinct values, which makes the aggregation expensive, as shown in §7.2. Probing a large hashtable with many collisions and updating its values cannot be done efficiently by SABER's single-threaded merge. LightSaber's parallel merge approach removes this bottleneck for such workloads.

In Fig. 13, we compare the scalability of LightSaber, SABER, and Scotty with and without parallel merge. For SG_2 and LRB_1, the parallel merge yields 3× and 2× higher throughput speedup, respectively. In contrast, SABER's performance is affected by its merge approach, which results

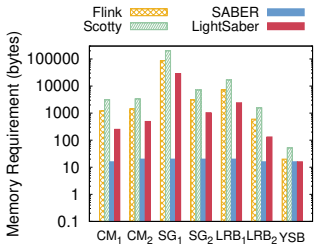


Figure 14: Memory consumption

in it being outperformed by Scotty for SG_2 after 5 cores. Although Scotty exhibits good scaling, it is consistently more than 6× worse compared to LightSaber, revealing the overhead of Flink's runtime [49].

7.6 Memory consumption

In Fig. 14, we evaluate the memory consumption for the different systems. Apart from the memory required for storing partial aggregates, we also consider the metadata as in previous work [69]. Flink stores an aggregate and the start/end timestamps per active window; Scotty with lazy slicing [69] maintains slices that require more metadata used for out-of-order processing; LightSaber only stores the required partial aggregates for the slices along with the maximum timestamp seen so far and a counter, as it operates on deterministic windows. This results in at least 3× and 7× less memory compared to Flink and Scotty, respectively.

On the other hand, SABER accesses tuples from the input stream directly, thus having three orders of magnitude lower memory consumption. Without slicing over the input stream, LightSaber can adopt this approach, as shown in Fig. 8f. This is more computationally expensive, however, because it requires repeated applications of the `inverse` combiner, leading to worse performance (see §7.2).

7.7 Evaluation of GAG

In this section, we explore the efficacy of GAG for both single- and multi-query workloads using the SM dataset. To evaluate different aggregation algorithms in an isolated environment, we run our experiments as a standalone process.

Each algorithm maintains sliding windows with a slide of 1 tuple by performing an eviction, insertion, and result production, which incurs a worst-case cost. We compare GAG to (i) SlickDeque (for non-invertible functions, we use a fixed size deque to get better performance); (ii) TwoStacks (using prior optimizations [66]); (iii) SlideSide; (iv) FlatFAT; and (v) SoE when applicable (e.g., for invertible functions). We evaluate the aforementioned algorithms in terms of throughput, latency, and memory requirements (in terms of partial aggregates to be maintained).

Single-query. For this experiment, the query computes a sum of an energy measure over windows with variable window sizes between 1 and 4 M tuples. As Fig. 15a shows, GAG behaves as SoE, exhibiting a throughput that is up to 1.4× higher than SlickDeque, because it avoids unnecessary conditional branch instructions.

For the non-invertible functions, we use min with the same window sizes as before. Fig. 15b shows that GAG has an up to 1.3× higher throughput compared to TwoStacks, due to the more efficient generated code, and 1.7× higher than SlickDeque, given its more cache-friendly data layout.

For a fixed window size of 16 K tuples and slide 1, we measure the latency for the SM_{sum} and SM_{min} queries. We omit results that exhibit identical performance (TwoStacks) or latency that is one order of magnitude higher (FlatFAT). Fig. 15c shows that our approach exhibits the lowest latency in min, max, average, and the 25^{th} and 75^{th} percentiles. This result is justified since GAG generates the most efficient code and removes the interpretation overhead in both cases.

Multi-query. In these experiments, we generate multiple queries with uniformly random window sizes in the range of [1, 128K] of tuples). The window slide for all queries is 1, which constitutes a worst case. We create workloads with 1 to 100 concurrent queries. For invertible functions, we use SM_{sum} and, for non-invertible ones, we use SM_{min}. For TwoStacks and SoE, we replicate their data structures for each window definition, because they cannot be used to evaluate multiple queries.

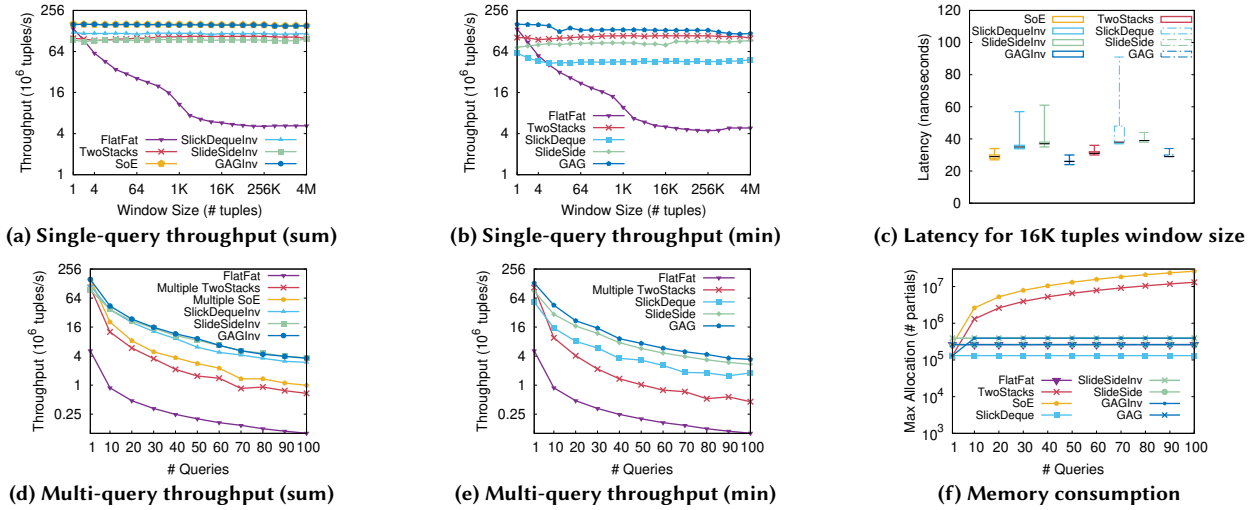
For invertible functions shown in Fig. 15e, GAG has comparable performance to SlideSide and outperforms SlickDeque by 45%. In Fig. 15e, we show that GAG for non-invertible functions outperforms SlideSide by 1.3× and SlickDeque by 2.7×, because it handles updates more efficiently.

In terms of memory consumption (see Fig. 15f), GAG maintains 3× more partial aggregates than SlickDeque for multiple invertible functions, similar to SlideSide. With non-invertible functions, GAG requires the same number of partial aggregates as FlatFAT and SlideSide, which is 2× more compared to SlickDeque. Note that for non-invertible functions, SlickDeque can use less memory with a dynamically resized deque, incurring a 2× performance degradation.

In summary, GAG generates code that achieves the highest throughput and lowest latency in all scenarios. For multi-query workloads, our approach trades-off performance with memory by requiring at most 3× more partials compared to the next best performing approach. Based on the benchmark queries from above, however, the number of partials is in the order of hundreds.

8 Related Work

Centralised streaming systems, such as STREAM [6], TelegraphCQ [20], and Niagara- CQ [21], have existed for decades but operate only on a single CPU core. More recent systems, such as Esper [26], Oracle CEP [53], and Microsoft StreamInsight [37], take advantage of multi-core CPUs at the



(a) Single-query throughput (sum)
(b) Single-query throughput (min)
(c) Latency for 16K tuples window size
(d) Multi-query throughput (sum)
(e) Multi-query throughput (min)
(f) Memory consumption

Figure 15: Comparison of incremental processing techniques

expense of weaker stream ordering guarantees for windows. S-Store [18] offers strong window semantics for SQL-like queries, but does not perform parallel window computation. StreamBox [50] handles out-of-order event processing and BriskStream [76] utilizes a NUMA-aware execution plan optimization paradigm in a multi-core environment. SABER [39] is a hybrid streaming engine that, in addition to CPUs uses GPUs as accelerators. These approaches are orthogonal to ours and can be integrated for further performance improvement. Trill [19] supports expressive window semantics with a columnar design, but it does not support the window aggregation approaches of LightSaber.

Distributed streaming systems, such as Spark [10], Flink [5], SEEP [17], and Storm [68], follow a distributed processing model that exploits the data-parallelism on a shared-nothing cluster. These systems are designed to account for issues found in distributed environments, such as failures [48, 56, 73], distributed programming abstractions [4, 10, 51], and efficient remote state management [15]. Millwheel [3] supports rich window semantics, but it assumes partitioned input streams and does not compute windows in parallel.

Window aggregation. Recent work on window aggregation [9, 13, 61–64, 67] has focused on optimizing different aspects of incremental computation. Instead of alternating between different solutions, with GAGs we generalize existing approaches and exhibit robust performance across different query workloads. Our work focuses on in-order stream processing, and we defer the handling of out-of-order algorithms, such as FiBA [65], to future work. Panes [45], Pairs [41], Cutty [16], and Scotty [69] are different slicing techniques, which are complementary to our work—LightSaber can generate code to support them. Leis et al. [44] propose a general algorithm for relational window operators by utilizing intra-partition parallelism for large hash groups and a specialized data structure for incremental computation. However, this work does not exploit the parallelism and incremental computation opportunities of window aggregation as LightSaber.

9 Conclusion

To achieve efficient window aggregation on multi-core processors, stream processing systems need to be designed to exploit both parallelism as well as incremental processing opportunities. However, we found that no state-of-the-art system exploits both of these aspects to a sufficient degree. Consequently, they all leave orders of magnitude of performance on the table. To address this problem, we developed a formal model of the stream processor design space and used it to derive a design that exploits parallelism as well as incremental processing opportunities.

To implement this design, we developed two novel abstractions, each addressing one of the two aspects. The first abstraction, *parallel aggregation trees (PATs)*, encodes the trade-off between parallel and incremental window aggregation in the execution plan. The second abstraction, *generalised aggregation graphs (GAGs)*, captures different incremental processing strategies and enables their translation into executable code. By combining GAG-generated code with the parallel execution strategy captured by the PAT, we developed the LightSaber streaming engine. LightSaber outperforms state-of-the-art systems by at least a factor 2 on all of our benchmarks. Some benchmarks even show improvement beyond an order of magnitude.

Acknowledgments

The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) is gratefully acknowledged.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[2] adamax. Re: Implement a queue in which `push_rear()`, `pop_front()` and `get_min()` are all constant time operations.. http://stackoverflow.com/questions/4802038. Last access: 11/04/20.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Is, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fer Andez-Moctezuma, Reuven Lax, Sam Mcveety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle Google. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Vldb* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[5] Apache Flink. https://flink.apache.org. Last access: 11/04/20.

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. 2003. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.* 26, 1 (March 2003), 19–26. http://sites.computer.org/debull/A03mar/paper.ps

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal* (June 2006). https://www.microsoft.com/en-us/research/publication/the-cql-continuous-query-language-semantic-foundations-and-query-execution/

[8] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 480–491.

[9] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-Window Aggregates. *Technical Report* 2004-15 (2004), 336–347. https://doi.org/10.1016/B978-012088469-8.50032-2

[10] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[11] Cagri Balkesen and Nesime Tatbul. 2011. Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In *Proceedings of the 8th International Workshop on Data Management for Sensor Networks (DMSN '11)*.

[12] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. 1987. FAD, a Powerful and Simple Database Language. In *VLDB*.

[13] Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. 2014. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware '14)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2663165.2663334

[14] Guy E. Blelloch. 1990. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA.

[15] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[16] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)*. ACM, New York, NY, USA, 1201–1210. https://doi.org/10.1145/2983323.2983807

[17] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 725–736. https://doi.org/10.1145/2463676.2465282

[18] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley Zdonik. 2014. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1633–1636. https://doi.org/10.14778/2733004.2733048

[19] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[20] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 668–668. https://doi.org/10.1145/872757.872857

[21] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Rec.* 29, 2 (May 2000), 379–390. https://doi.org/10.1145/335191.335432

[22] Xin Chen, Charng-Da Lu, and K. Pattabiraman. 2014. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE '14)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 167–177. https://doi.org/10.1109/ISSRE.2014.34

[23] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[24] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 339–350.

[25] N. Elmqvist and J. Fekete. 2010. Hierarchical Aggregation for Information Visualization: Overview, Techniques, and Design Guidelines. *IEEE Transactions on Visualization and Computer Graphics* 16, 3 (May 2010), 439–454. https://doi.org/10.1109/TVCG.2009.84

[26] Esper. http://www.espertech.com/esper/. Last access: 11/04/20.

[27] 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

[28] Jamie Grier. 2016. Extending the Yahoo! Streaming Benchmark. https://www.ververica.com/blog/extending-the-yahoo-streaming-

benchmark. Last access: 11/04/20.

[29] 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

[30] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

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

[32] Intel. 2020. Intel threading building blocks. https://software.intel.com/en-us/intel-tbb. Last access: 11/04/20.

[33] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philipe Selo, and Chitra Venkatramani. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *25th ACM SIGMOD International Conference on Management of Data (SIGMOD '06)* (25th acm sigmod international conference on management of data (sigmod '06) ed.). ACM. https://www.microsoft.com/en-us/research/publication/design-implementation-evaluation-linear-road-benchmark-stream-processing-core/

[34] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 393–398. https://doi.org/10.1145/2335484.2335536

[35] 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

[36] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. 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)*. IEEE Computer Society, Washington, DC, USA, 94–103. https://doi.org/10.1109/CCGRID.2010.112

[37] Seyed Jalal Kazemitabar, Ugur Demiryurek, Mohamed Ali, Afsin Akdogan, and Cyrus Shahabi. 2010. Geospatial Stream Query Processing Using Microsoft SQL Server StreamInsight. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1537–1540. https://doi.org/10.14778/1920841.1921032

[38] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[39] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA.

[40] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[41] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly Sharing for Streamed Aggregation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 623–634. https://doi.org/10.1145/1142473.1142543

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[43] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[44] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient processing of window functions in analytical SQL queries. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1058–1069. https://doi.org/10.14778/2794367.2794375

[45] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44. https://doi.org/10.1145/1058150.1058158

[46] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 311–322. https://doi.org/10.1145/1066157.1066193

[47] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 274–288. https://doi.org/10.14778/1453856.1453890

[48] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 439–453. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/lin

[49] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[50] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 617–629. http://dl.acm.org/citation.cfm?id=3154690.3154749

[51] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[52] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[53] Oracle®Stream Explorer. http://bit.ly/1L6tKz3. Last access: 11/04/20.

[54] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and Georgios Theodorakis. 2018. Do We Need Distributed Stream Processing? https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. Last access: 11/04/20.

[55] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1707–1718.

[56] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2465351.

2465353

[57] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, and et al. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[58] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Acm Sigplan Notices*, Vol. 46. ACM, 127–136.

[59] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1907–1922.

[60] Ambuj Shatdal and Jeffrey F. Naughton. 1995. Adaptive Parallel Aggregation Algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 104–114. https://doi.org/10.1145/223784.223801

[61] A.U. Shein, P.K. Chrysanthis, and A. Labrinidis. 2017. FlatFIT: Accelerated incremental sliding-window aggregation for real-time analytics. *ACM International Conference Proceeding Series* Part F1286 (2017). https://doi.org/10.1145/3085504.3085509

[62] 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

[63] Kanat Tangwongsan and Martin Hirzel. 2015. General Incremental Sliding-Window Aggregation. *Pvldb* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940

[64] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. 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* (2017), 66–77. https://doi.org/10.1145/3093742.3093925

[65] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2019. Optimal and General Out-of-Order Sliding-Window Aggregation. *Proc. VLDB Endow.* 12, 10 (June 2019), 1167–1180. https://doi.org/10.14778/3339490.3339499

[66] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2018. Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation, See [66], 34–41. http://www.adms-conf.org/2018-camera-ready/SIMDWindowPaper_ADMS%2718.pdf

[67] Georgios Theodorakis, Peter R. Pietzuch, and Holger Pirk. 2020. SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries. In *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 435–438. https://doi.org/10.5441/002/edbt.2020.51

[68] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[69] 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

[70] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3132747.3132750

[71] Stratis D. Viglas and Jeffrey F. Naughton. 2002. Rate-based Query Optimization for Streaming Information Sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/564691.564697

[72] John Wilkes. 2011. More Google Cluster Data. Google Research Blog, http://bit.ly/1A38mfR. Last access: 11/04/20.

[73] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[74] Erik Zeitler and Tore Risch. 2011. Massive Scale-out of Expensive Continuous Queries. *PVLDB* 4, 11 (2011), 1181–1188. http://www.vldb.org/pvldb/vol4/p1181-zeitler.pdf

[75] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 516–530. https://doi.org/10.14778/3303753.3303758

[76] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 705–722. https://doi.org/10.1145/3299869.3300067