Scalable and Fault-Tolerant Data Stream Processing on Multi-Core Architectures

Georgios Rafail Theodorakis

Department of Computing
Imperial College London

This dissertation is submitted for the degree of
Doctor of Philosophy

May 2023

I dedicate this thesis to my parents, Giannis and Erietta.

Declaration

This thesis presents my work in the Department of Computing at Imperial College London between October 2017 and October 2021. Parts of the work described were done in collaboration with other researchers:

- **Chapter 3**: I proposed, implemented and evaluated the design of the LIGHTSABER stream processing engine. The system's design resulted from many fruitful discussions with Alexandros Koliousis, Holger Pirk, and Peter Pietzuch.
- **Chapter 4**: I led the development and evaluation of the SCABBARD engine. The choice of compression algorithms resulted from discussions with Fotios Kounelis. Fotios also contributed to the compression scheme's evaluation as part of his Ph.D. project at Imperial College London during the academic year 2020-2021.

I declare that the work presented in this thesis is my own, except where declared above.

Georgios Rafail Theodorakis
May 2023

Copyright

Georgios Rafail Theodorakis
May 2023

Related Publications

The following peer-reviewed publications have directly contributed to this thesis and are discussed in detail:

- **Scabbard: Single Node Fault-Tolerant Stream Processing.**
  Georgios Theodorakis, Fotios Kounelis, Peter Pietzuch, Holger Pirk.
  48th International Conference on Very Large Data Bases (**VLDB**), Sydney, Australia, 2022 [206]

- **LightSaber: Efficient Window Aggregation on Multi-core Processors.**
  Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, Holger Pirk.
  ACM International Conference on Management of Data (**SIGMOD**), Portland, OR, USA, 2020 [205]

- **SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries.**
  Georgios Theodorakis, Peter Pietzuch, Holger Pirk.
  23rd International Conference on Extending Database Technology (**EDBT**), Copenhagen, Denmark, 2020 [207]

- **Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation.**
  Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, Holger Pirk.
  9th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (**ADMS**), Rio de Janeiro, Brazil, 2018 [204]

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisors, Prof. Peter Pietzuch and Dr. Holger Pirk, for their continuous support over the past five years. This thesis would not have been possible without their passion for research, encouragement, mentorship at different levels, and high standards.

I want to thank Dr. Alexandros Koliousis, who helped me in countless ways during my first PhD years. Our conversations guided me to become a better person and researcher. I am also grateful to my examiners, Asterios Katsifodimos and Alastair Donaldson, for their insightful suggestions that helped improve this document.

During my studies, I enjoyed meeting and collaborating with people from various research groups and making friends on campus and beyond. First, I would like to thank Imperial College London and the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) for supporting my studies. I also want to thank my friends and colleagues from the Large-Scale Data and Systems (LSDS) Group and the HiPEDS programme: Georgios, Dan Iorga, Jukka, Alexandros Tasos, Dom, Christian, Joshua, Huanzhou, Jana, Luo, Dan O'Keeffe, Pierre-Louis, Simon, Ahmad, Alessandro, Konstantinos, Marcel, Alex, Daphne, Shujie, and Lluís. Special thanks to Panagiotis, Fotis, and Hubert for all work done together. Our technical and off-topic discussions made these past years a pleasant experience. Next, I want to thank Paris, Max, and the other Continuous Deep Analytics (CDA) group members at KTH for their invaluable advice and technical expertise.

I would also like to thank Max Heimel and the Snowflake team in Berlin as well as Nikos Ntarmos and the Huawei R&D team in Edinburgh for the opportunities to intern with them. Both internships were inspirational and memorable experiences for my PhD studies.

Thanks to my wonderful friends from London and back home for supporting me on the good and bad days. I will always be grateful to Evgenia for keeping me sane and being there for me from the beginning of this journey.

Finally, I am incredibly grateful to my family, who inspired and supported me throughout these years. I thank my parents, Giannis and Erietta, and my sister Marianna for their endless love and encouragement.

Abstract

With increasing data volumes and velocity, many applications are shifting from the classical "process-after-store" paradigm to a stream processing model: data is produced and consumed as continuous streams. Stream processing captures latency-sensitive applications as diverse as credit card fraud detection and high-frequency trading. These applications are expressed as queries of algebraic operations (e.g., aggregation) over the most recent data using windows, i.e., finite evolving views over the input streams. To guarantee correct results, streaming applications require *precise window semantics* (e.g., temporal ordering) for operations that maintain state.

While high processing throughput and low latency are performance desiderata for stateful streaming applications, achieving both poses challenges. Computing the state of overlapping windows causes redundant aggregation operations: incremental execution (i.e., reusing previous results) reduces latency but prevents parallelization; at the same time, parallelizing window execution for stateful operations with precise semantics demands ordering guarantees and state access coordination. Finally, streams and state must be recovered to produce consistent and repeatable results in the event of failures.

Given the rise of shared-memory multi-core CPU architectures and high-speed networking, we argue that it is possible to address these challenges in a single node without compromising window semantics, performance, or fault-tolerance. In this thesis, we analyze, design, and implement stream processing engines (SPEs) that achieve high performance on multi-core architectures. To this end, we introduce new approaches for in-memory processing that address the previous challenges: (i) for overlapping windows, we provide a family of window aggregation techniques that enable *computation sharing* based on the algebraic properties of aggregation functions; (ii) for parallel window execution, we balance parallelism and incremental execution by developing abstractions for both and combining them to a novel design; and (iii) for reliable single-node execution, we enable strong *fault-tolerance* guarantees without sacrificing performance by reducing the required disk I/O bandwidth using a novel persistence model. We combine the above to implement an SPE that processes hundreds of millions of tuples per second with sub-second latencies. These results reveal the opportunity to reduce resource and maintenance footprint by replacing cluster-based SPEs with single-node deployments.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In the Internet of Things (IoT) era, billions of network-connected devices [111] produce massive amounts of data ready for analysis [186]. As data is now cheap to generate and store [154], large organizations and internet companies, such as Amazon, Google, and Microsoft, have shifted their focus to scalable data processing technologies. The explosive growth of *big data* [174] volumes and velocity has fueled the development of several processing systems that enable extracting valuable information and making critical decisions. Performing *big data* analysis efficiently allows our society to accelerate its progress in a wide variety of domains, ranging from finance to healthcare and environmental issues.

Along with the growth of *big data*, we also witness the emergence of a new type of long-running applications [21, 86] that execute continuously as new data arrives in the form of (possibly infinite) streams. According to recent estimates [186], by 2025, 30% of all data will be analyzed in real-time. Therefore, it is not surprising that the stream processing paradigm (i.e., consuming and producing data as streams) is established as the fourth important data-intensive workload [206], next to transaction processing, reporting, and online analytics.

In contrast to existing solutions [69, 230] that focus on offline batch processing (i.e., the classic "process-after-store" model), streaming applications require low end-to-end processing latency results in addition to high processing throughput. Examples of such applications are high-frequency trading [166, 197], clickstream analytics [6, 49, 95], credit card fraud detection [79], live sensor monitoring [61, 62], social network analysis [54], targeted advertising [6], and real-time visualization [209]. The common characteristic of these latency-sensitive applications is the extraction of knowledge from data streams to perform *time-critical decisions* in *sub-second latencies*. Delaying responses in such scenarios either reduces their value or results in adverse effects (e.g., health monitoring requires immediate actions) because the significance of each input tuple decays over time.

As big data volumes and velocity grow continuously, and streaming applications depend on more rigorous performance requirements, there is a demand for efficient real-time data-parallel analytics on modern hardware. Existing stream processing engines (SPEs) focus on solving the inherent problems of distributed and fault-tolerant stream processing [6, 7, 45, 163, 231] on commodity clusters [196]. This leads to a lack of performance guarantees, approximate window semantics[1], and high resource and maintenance overheads. At the same time, the rise of shared-memory multi-core CPU architectures and high-speed networking has established single-node SPEs as a reasonable alternative approach [137, 157, 158, 236]. Therefore, our idea is to provide a novel design for relational SPEs: a single-node deployment that addresses the challenges of stateful data processing, while achieving the same performance requirements as distributed SPEs, without compromising fault-tolerance or application semantics.

1.1 Towards hardware-efficient data stream processing

Over the last two decades, data stream processing has been an active research field that evolved from databases and distributed systems [85]. Stream processing enables continuous applications [21] represented as a set of operators that apply computation on infinite stream sequences and output streams. These operators can be *stateless* or *stateful* depending on whether they have to remember previous data (i.e., state) using *windows*.

Windows allow streaming operators to apply algebraic operations (e.g., aggregations or joins) on the most recent data by constructing finite views over unbounded streams. These (potentially overlapping) views evolve over time and can be regarded as temporary relations [169]. To guarantee consistent and correct results, window-based applications must account for *precise window semantics* [169]: constructing windows based on a windowing attribute that establishes the order among data [147]. More specifically, the attribute determines when data becomes visible for processing (i.e., when to create a new window and trigger computation). The most common windowing attributes are physically (i.e., event time) or logically attached timestamps (i.e., ingestion time [7]) to input tuples. For example, time-based windows rely on the input's temporal ordering. With ever-growing data volumes and velocities, the importance of window-based real-time analysis has increased.

To accommodate such data amounts, we have witnessed several distributed SPEs [45, 231, 163, 213, 6, 80, 120, 44] that rely on shared-nothing clusters of commodity machines [2]

[1] Early distributed SPEs [231, 211] lack native support for event time windows [7] (i.e., windows based on timestamps attached to input tuples).

[2] A shared-nothing architecture [196] is a distributed architecture that eliminates contention among nodes by independently accessing memory or storage.

to scale complex window-based analytics over massive data streams. These commodity machines constitute the processing nodes (physical or virtual machines) that host fragments of stream queries. Many research groups from both academia [20, 55, 80, 231, 45] and internet companies [6, 120, 163, 49] have shaped the design of modern SPEs into sophisticated and scalable engines, which produce correct results in the event of non-deterministic software or hardware failures [134] (e.g., a node crashing).

Although current distributed SPE designs cover a wide range of real-world use cases and achieve high processing throughput with a scale-out model, they face significant challenges when performing parallel stateful computations. First, their design introduces cross-process and network communication overheads (e.g., serialization) and assumes the availability of compute clusters, especially for fault-tolerance. This results in an increased resource and maintenance footprint. Furthermore, their designs exhibit unpredictable performance guarantees due to the network communication latency of multiple nodes (e.g., TCP/IP latency spikes). Finally, distributed SPEs struggle to provide efficient parallelization strategies for precise window semantics: they either perform redundant computations [137] or approximate window semantics due to the lack of a global clock in a distributed environment.

As a result, existing SPEs are not suitable for supporting a set of growing time-sensitive applications, such as hospital medical monitoring, abnormally trajectory detection [235], and high-frequency trading [166]. These applications require low end-to-end latency with predictable spikes over vast data streams that distributed execution cannot guarantee. For example, as recent analysis shows [166], a single stock symbol stream may reach up to millions of messages per second with microseconds median latency requirements.

Given the emergence of highly-parallel scale-up architectures (i.e., multi-core CPUs and GPUs) and modern network technologies in data centers and clouds [64], there is a demand for new software solutions. Such solutions must exploit current hardware trends (e.g., non-uniform memory access, SIMD/multi-core parallelism, and terabytes of memory [216]) to provide more predictable performance guarantees. In addition, high-speed networking such as Remote Direct Memory Access (RDMA) [125, 36, 41] makes scale-up designs practical for many latency-critical scenarios, because it allows for fast stream ingestion and remote storage [135].

As a response to this demand, many scale-up designs for single-node SPEs were developed as academic prototypes [137, 158, 236, 157] or enterprise solutions [49]. Modern single-node SPEs achieve higher performance with fewer resources and costs by avoiding abstractions for distributed processing [175]. However, existing scale-up designs implement ad-hoc aggregation and parallelization strategies, which achieve high performance only for specific queries depending on the window specification and aggregation function's algebraic

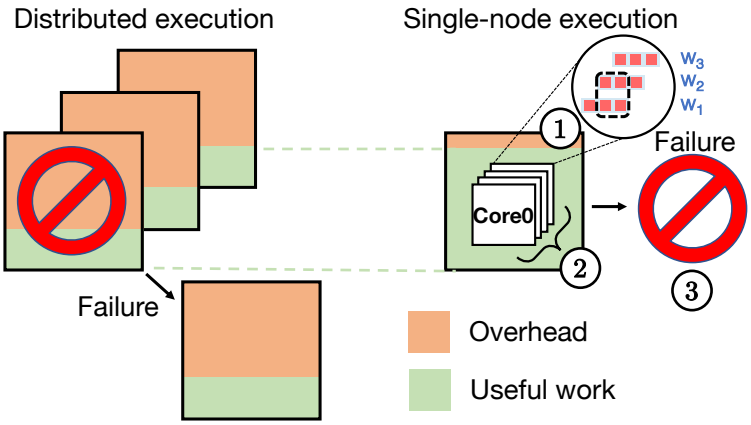


Fig. 1.1 Challenges for scalable and reliable single-node stream processing

properties. In addition, they neglect single-node fault-tolerance as existing fault-tolerance mechanisms cannot handle their high query performance due to limited available resources (e.g., disk I/O bandwidth).

Therefore, we identify a unique opportunity for revisiting the design of a hardware-efficient SPE on multi-core CPU architectures. The rationale behind such a solution is twofold: (i) supporting time-sensitive applications with predictable guarantees despite increasing data volumes and velocity; and (ii) optimizing resource utilization of stream processing on a single node. By achieving the latter, SPEs can also accelerate distributed execution, as the footprint of processing nodes to achieve the same performance is reduced. While single-node deployments pose physical limitations in terms of available resources, most streaming applications' memory requirements are in the order of GBs. For example, Facebook's processing ecosystem [54] handles hundreds of GB/s, while high-frequency trading [166] or fraud detection applications [166] ingest hundreds of millions of tuples per second during load peaks.

Yet, designing a *general-purpose* relational SPE that can transparently take advantage of existing hardware while executing arbitrary streaming *windowed* [21] SQL queries remains an open challenge. For such applications, it is necessary to provide precise window semantics without compromising performance requirements (i.e., throughput and latency) or fault-tolerance. Figure 1.1 illustrates the main challenges for a single-node SPE design. For *hardware-efficient* single-core execution (1), it is crucial to provide *computation sharing* for window operators, which capture diverse streaming workloads not expressed by relational operators. For the multi-core *operator parallelization* (i.e., *scalability*) (2), the proposed strategy must account for precise window ordering guarantees and state access coordination. Finally, while current *fault-tolerance* mechanisms handle failures in a distributed environment, the high query performance of single-node designs makes them impractical for scale-up

scenarios. There is, thus, a demand for novel single-node *fault-tolerance* mechanisms that provide correct results despite the limited available resources (3). Let us now outline the motivation of this thesis and discuss the three challenges in more detail.

1.2 Research motivation

To support the continuously growing data volumes, velocity, and performance requirements of real-time analytics, there is a need to design scalable and fault-tolerant SPEs that efficiently utilize modern hardware. Existing SPEs already scale streaming applications on shared-nothing clusters of commodity machines reliably. However, the precise window semantics and strict latency requirements of a wide range of *time-sensitive* applications demand more sophisticated designs and abstractions on multi-core CPU architectures. As a response, in this thesis, we argue that *it is possible to address the scalability and fault-tolerance challenges of stream processing in a single node and avoid the expensive scale-out approach*. We provide a novel solution that exploits today's computing infrastructure in data centers and clouds to accelerate scale-up performance.

As shown in Figure 1.1, there are three fundamental mismatches between the demands of modern streaming applications and existing SPE designs:

- **Lack of CPU-efficient computation sharing for window aggregation.** Regarding *hardware-efficient single-core execution* (1), we focus on calculating windowed aggregations [79, 95, 49, 6], a core streaming operator that performs redundancy-prone computations, especially for windows with overlapping content. Use cases of overlapping windows arise from either sliding window definitions (i.e., windows that slide over data based on a specified interval smaller than the window size) or multiple queries running concurrently over the same input stream. Existing approaches perform redundant operations (affecting latency) or incremental execution (limiting throughput and scalability). This results in significant execution overheads for SPEs [212, 40, 199] and leads to expensive scale-out approaches.

- **Lack of efficient parallelization strategies for window operators.** Following the implementation of hardware-efficient operators, a modern SPE design must consider the *parallel window execution strategy* (2) on tens of CPU cores in a non-uniform-memory-access (NUMA) architecture[3] while preserving precise window semantics. However, modern SPEs face a trade-off for windowed operators between exploiting incremental execution (i.e., reusing previous results) and instruction/multi-core-level

[3] It is more expensive for a processor to access memory owned by or shared with another processor.

parallelism (i.e., aggregation can be executed with either approach but not both). Therefore, they incorporate ad-hoc solutions that achieve high performance only for a limited range of applications and yield inconsistent performance based on the workload characteristics (i.e., the window definition or the type of aggregation functions).

- **Lack of efficient fault-tolerant techniques for single-node scale-up SPEs.** Although single-node multi-core SPEs can process hundreds of millions of tuples per second, for an SPE to be practical for real-world scenarios, it is crucial to provide *efficient fault-tolerant mechanisms* (3). However, making single-node SPEs fault-tolerant with exactly-once semantics without performance loss is an open challenge. Due to the limited I/O bandwidth of a single-node SPE, it becomes infeasible to persist all stream data and operator state (i.e., checkpointing) during execution. Thus, single-node SPEs rely on distributed systems, such as Apache Kafka [16], to recover stream and operator data after a failure, necessitating complex cluster-based deployments. This lack of built-in fault-tolerance features limits the performance of existing SPEs and hinders the adoption of single-node deployments.

1.3 Research contributions

This thesis tackles the aforementioned challenges in single-node deployments without compromising window semantics, performance requirements, and fault-tolerance. To address the challenge of CPU-efficient window aggregation, we introduce two novel aggregation techniques for data-parallel (using SIMD vector instructions) and incremental execution. These approaches improve performance by: (i) accelerating the computation of the partial aggregates (i.e., aggregations over non-overlapping parts of a window discussed in Section 2.1) through hardware-conscious optimizations; and (ii) increasing the result-sharing efficiency between multiple aggregate queries using their algebraic properties. To satisfy the window operator parallelization challenge, we introduce LIGHTSABER, a state-of-the-art SPE that balances parallelism and incremental processing for windowed stream queries using two novel abstractions as well as JIT query compilation for efficient multi-core execution. Finally, we introduce SCABBARD to provide strong fault-tolerance semantics in single-node deployments despite the limited resources. SCABBARD provides novel persistence abstractions and adaptive compression techniques (i.e., encoding data with fewer bits than its original representation) to enable reliable stream processing without compromising performance. By combining the above contributions, we design an SPE that outperforms state-of-the-art systems in throughput while providing sub-second latencies and poses a practical alternative to scale-out models with fewer resources. Let us now present these contributions in detail.

1.3.1 Efficient window aggregation and multi-query sharing

As discussed in Section 1.2, window aggregation is a core operation for streaming analytics that introduces redundancy-prone computations in two cases: (i) when computing sliding windows; and (ii) when answering concurrent queries that share intermediate aggregate results. To tackle the first class of workloads, we introduce HammerSlide, a novel technique that accelerates partial window aggregation by extending existing algorithms [106] for CPU-efficient execution and providing data-level parallelization. When integrated with a state-of-the-art multi-core SPE [137], HammerSlide yields up to 12× greater throughput than the baseline. Regarding the second class of workloads, we introduce SlideSide. This novel algorithm increases the sharing opportunities of intermediate aggregate results between concurrent aggregation queries based on their algebraic properties, yielding up to 2.2× greater throughput than existing solutions with comparable latency.

1.3.2 Scaling window operators on multi-core processors

For the parallelization of window operators, we first describe a general model for the design space of window aggregation strategies that captures existing design decisions and enables reasoning about entirely new ones. Based on this model, we introduce LIGHTSABER, a new SPE that balances parallelism and incremental processing when executing window aggregation queries on multi-core CPU architectures. Its design generalizes and extends existing approaches: (i) for *parallel* processing, LIGHTSABER exploits the parallelism of modern processors by constructing an *aggregation tree*. This tree divides window aggregation into intermediate steps that enable the efficient use of both data-level (i.e., SIMD) and task-level (i.e., multi-core) parallelism; and (ii) to generate efficient *incremental* code from the aggregation tree, LIGHTSABER uses a graph data structure, which encodes the low-level data dependencies required to produce aggregates over the stream. Therefore, the graph generalizes state-of-the-art approaches for incremental window aggregation introduced in the previous contribution, supports work-sharing between overlapping windows, and adapts to workload characteristics (e.g., algebraic properties or number of queries). LIGHTSABER achieves up to an order of magnitude higher throughput compared to existing systems — on a 16-core server, it processes 470 million tuples/s with 132 μs median latency. Having introduced our parallelization strategy for window aggregation, let us now focus on the mechanisms required for single-node fault-tolerance.

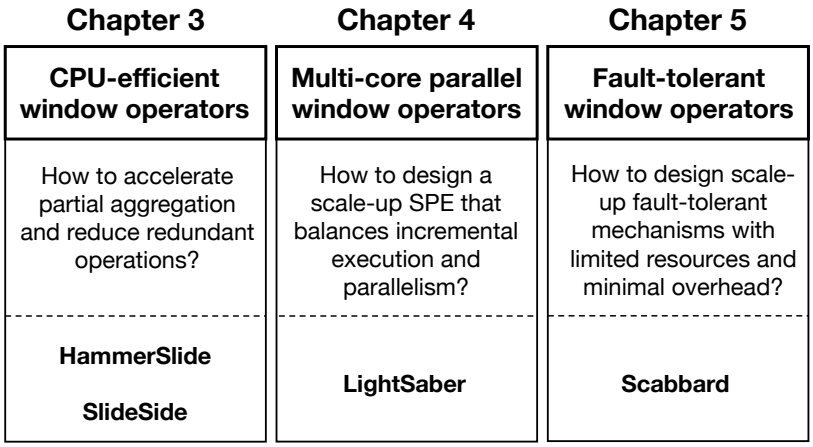


Fig. 1.2 Mapping challenges to contributions in thesis' chapters

1.3.3 Efficient single-node fault-tolerant stream processing

For *reliable* stream processing, we describe SCABBARD, the first single-node SPE supporting exactly-once fault-tolerance semantics despite limited local I/O bandwidth. SCABBARD reduces the required disk I/O bandwidth by tightly integrating *stream* and *state persistence* with query execution using a novel graph representation for the streaming operators. This representation enables both compile-time and runtime optimizations. Within the operator graph, SCABBARD determines when to persist streams based on the operators' selectivity (i.e., tuples produced for every tuple arrived as input): persisting streams after operators discard data can substantially reduce the required I/O bandwidth. As part of the operator graph, SCABBARD supports parallel persistence operations and uses markers to decide when to discard persisted data. The persisted data volume is further reduced using workload-specific compression: SCABBARD monitors stream statistics and dynamically generates computationally efficient compression operators based on the data characteristics. Our experiments show that SCABBARD can execute stream queries that process over 200 million tuples per second while recovering from failures with sub-second latencies.

1.4 Dissertation outline

Figure 1.2 illustrates the mapping between the challenges we focus on in this thesis (Section 1.2) and our contributions (Section 1.3). The remainder of this thesis is structured as follows:

- Chapter 2 presents background and related work. It first describes basic concepts and terminology for stream processing and then traces the evolution of SPEs with an

emphasis on hardware-conscious designs. The chapter then introduces background related to modern processors, storage, and the network stack found in data centers. It discusses the underlying concepts behind window aggregation while comparing existing solutions to showcase their limitations in terms of efficient partial aggregation and intermediate result sharing. Subsequently, the chapter identifies the limitations of current solutions for parallelizing window operators, which do not exhibit robust performance across different workloads, through a series of experiments. Finally, it examines how SPEs provide fault-tolerance and demonstrates their limitations, especially when deployed in a single-node system.

- Chapter 3 describes HammerSlide and SlideSide, the two novel techniques proposed for CPU-efficient window aggregation. The chapter performs an in-depth analysis of existing solutions and presents an approach that addresses two individual challenges: (i) how to accelerate partial window aggregation given the complex data dependencies introduced with sliding windows; and (ii) how to efficiently share intermediate aggregate results, especially when answering multiple concurrent queries through incremental execution. It explains how HammerSlide addresses the first problem through hardware-conscious optimizations (i.e., SIMD) and shows how it achieves up to 12× better performance when integrated with an SPE. For the second problem, the chapter presents SlideSide, which achieves up to 2× better throughput when computing concurrent queries by exploiting their algebraic properties.

- Chapter 4 describes LIGHTSABER, a novel SPE that balances parallelism and incremental processing on multi-core CPUs. The chapter first explains a general aggregation model that captures existing strategies. Then, it focuses on how window aggregation can be parallelized efficiently using an abstraction we call *parallel aggregation tree* (PAT). The chapter discusses how LIGHTSABER generates code for streaming applications at a query level and for workload-aware incremental execution, in which case it uses an abstraction called *generalized aggregation graph* (GAG). Next, it presents how LIGHTSABER performs memory management and NUMA-aware execution. Finally, in a range of experiments, it shows that LIGHTSABER outperforms state-of-the-art systems, such as Apache Flink, by a factor of seven for standardized benchmarks [58], and up to one order of magnitude for other queries.

- Chapter 5 describes SCABBARD, a novel single-node fault-tolerant SPE built atop LIGHTSABER that provides exactly-once results without compromising performance. It first introduces the data and fault-tolerance models of this work. Then, it presents a new *persistent operator graph* model that allows SPEs to make *workload-aware*

decisions when persisting data. Next, the chapter explains how an SPE can perform query-specific adaptive compression through JIT code generation to accelerate persistence and how we combine the above to design SCABBARD. Finally, through experiments, it shows that SCABBARD introduces minimal overhead compared to a system without fault-tolerance while outperforming existing systems, such as Apache Flink, by at least one order of magnitude in throughput.

- Chapter 6 summarizes the contributions of the thesis and presents potential future research directions.

Chapter 2

Background

This chapter describes the basic concepts, requirements, and open challenges related to stream processing. First, it provides an intuition about the class of applications that we address in this work, introduces a list of requirements for designing an efficient SPE, and traces the evolution of SPEs (Section 2.1). It then provides a brief overview of the hardware components involved when designing a hardware-efficient scale-up SPE, with a focus on modern processors, the storage layer, and the network stack (Section 2.2). Subsequently, it covers the underlying concepts behind window aggregation, a core component of streaming analytics, and presents how existing approaches fall short of addressing the challenges of efficient single-core execution (Section 2.3). The chapter introduces the parallelization strategies of SPEs and showcases their limitations on multi-core execution (Section 2.4). Finally, it presents existing mechanisms for fault-tolerant stream processing and illustrates their limitations for single-node deployments (Section 2.5).

2.1 Characteristics and challenges of stream applications

According to estimates from recent studies [186], by 2025, the global datasphere will grow to 175 ZB, and nearly 30% is likely to be analyzed in real-time. This exponential data growth in volume and velocity encompasses new challenges for real-time applications aiming to extract valuable information for companies and organizations. Web applications [95, 49, 6], sensor networks [62, 61], location-based services, network traffic monitoring, and electronic trading [197] are only a few application examples that rely on data stream processing to perform analytics with high throughput and low end-to-end latency results. Data stream processing refers to the evaluation of continuous queries [21] over unbounded data streams. Let us now provide an example of a continuous application using the *continuous query language* (CQL) [21].

```
select timestamp, category, sum(cpu) as totalCpu
from TaskEvents [range 60 slide 1]
group by category
```

Fig. 2.1 CM_1 query in CQL

Example: The query in Figure 2.1 taken from the Cluster Monitoring (CM) [220] benchmark and emulates a monitoring application that computes the total CPU utilization of different task categories with tuples collected from a compute cluster at Google as input. As new data arrives, the query groups the input based on a grouping attribute (i.e., `category`), forms *time-based* windows of 60 seconds interval updated every second, applies a sum operation over them, and emits a new stream as an output.

Precise window semantics are important for such applications because they enable processing infinite input streams with (temporal) ordering guarantees by constructing windows over the most recent data (we explain windows comprehensively in Section 2.1.2). The emitted output can be used as the input of another query, persisted on disk for later analysis, or visualized using a dashboard [212]. Users can combine different relational operators [21] with windows, such as projection, selection, join, or aggregation, to specify continuous queries, perform complex analytics, and extract knowledge from the input streams. To execute such analytics at scale, users utilize distributed SPEs such as Apache Flink [15] or Apache Spark [17], which offer user-friendly programming interfaces and scale streaming computations transparently and reliably to shared-nothing clusters of commodity machines [196]. These systems focus on the inherent distributed stream processing problems [231, 163, 6] and neglect efficiency on multi-core CPU architectures.

Yet, the fast-paced growth of data volumes and velocity, along with the strict performance requirements of a wide range of emerging *time-critical* applications [235, 166] pose new challenges to distributed SPE designs. While various approaches to perform analytics with precise window semantics exist, we demonstrate experimentally how current solutions fail to satisfy modern application requirements in terms of CPU efficiency and parallelization in Sections 2.3 and 2.4. In addition, given that an SPE must deliver deterministic results upon failures [211, 6, 80, 24], in Section 2.5, we discuss how existing mechanisms exhibit limitations, especially in a single-node scale-up deployment. Therefore, we identify an opportunity to revisit the design of SPEs on multi-core CPU architectures.

This section is structured as follows. First, in Section 2.1.1, we briefly introduce basic application requirements that streaming algorithms and SPEs must satisfy. We then focus on

the design characteristics of SPEs (Section 2.1.2) before proceeding with their evolution and their limitations on exploiting modern hardware (Section 2.1.3).

2.1.1 Data stream processing requirements

Stream processing applications pose many unique challenges in the database community due to their continuous nature, which requires low latency results not supported by the classical "process-after-store" model. Examples of streaming problems are designing efficient algorithms, programming interfaces, and scalable and reliable engines to analyze continuous data. Throughout this thesis, we adopt the five following requirements introduced by Stonebraker et al. [197][1] when designing a streaming algorithm or a system component and discuss how they relate to the contributions from the following chapters:

1. **Process and respond instantaneously**: An SPE must utilize the available hardware resources efficiently to perform high throughput and low-latency computation. Our work exploits modern hardware, and we specifically focus on CPU-efficient execution, storage, and data transfer in Chapters 3, 4, and 5.

2. **Keep the data moving**: An SPE must process messages in-memory, avoiding costly memory allocation and storage operations in the critical process path, as we discuss in Chapters 3 and 4. Regarding efficient streaming algorithms, in Chapter 3, we contribute approaches for on-the-fly computation.

3. **Query using a high-level language on streams**: The support of a high-level language, such as the *continuous query language* (CQL) [21], is required, as it is capable of capturing the underlying complexity in a plethora of operations on data without exposing the details to the user. Our solution uses a lower-level imperative API influenced by CQL to support such operators, which is extensible by using the programming interfaces discussed in Chapter 4 and Chapter 5.

4. **Generate predictable results**: The results of an SPE must be deterministic and repeatable to be valuable without compromising window semantics, as we discuss in Chapter 5.

5. **Guarantee data safety and availability**: An SPE should support high availability services and ensure the integrity of data, even upon failures, as discussed in Chapter 5.

[1] We discuss in Section 6.2 how we can extend our work to support the remaining requirements.

2.1.2 The design characteristics of SPEs

In this section, we introduce five fundamental design aspects of any modern SPE: (i) the *data model*; (ii) the *dataflow graphs* that model application computation; (iii) *window mechanisms*; (iv) the types of *parallelism* for streaming execution; and (v) the streaming *execution models*. Next, in Section 2.1.3, we present the evolution of SPEs.

Data model. Following the semantics of the *continuous query language* (CQL) [21], we adopt a relational stream model with windows.

Definition 2.1.1 (Data stream). A data *stream* $s = \langle t_1, t_2, ... \rangle$ is an infinite sequence of *tuples*. Each tuple $t = (\varepsilon, p)$ has: an event timestamp $\varepsilon(t) \in \mathscr{E}$ that denotes when the event occurred, where $\mathscr{E}$ is an ordered time domain of discrete non-negative integer values; and p, a sequence of values of primitive data types.

There are two "types" of streams: the source data streams that arrive at the SPE, called *base streams*, and the streams produced by operators, called *derived streams*. Below, we describe how the *derived streams* are assigned timestamps based on window semantics.

Definition 2.1.2 (Temporal ordering). The order of tuples in a stream respects their event timestamps, referred to as the *monotonicity* property: $\forall\, t_i, t_j \in s$, *if* $i < j$, *then* $\varepsilon(t_i) \leq \varepsilon(t_j)$.

Temporal ordering influences window semantics in the following aspects: (i) window construction becomes deterministic when using event timestamps; (ii) operator implementations become simplified, as there is no need to buffer and re-order tuples; (iii) no additional logic is required to handle data that arrive late (see punctuations below).

The dataflow graph. SPEs model stream processing applications as directed operator graphs [65], with operators as vertices and data (streams) flowing between operators as edges. In most cases, streaming application graphs are considered acyclic, and streams are implemented as FIFO queues. At a specific time instant $\varepsilon(t)$, a queue (i.e., stream) contains a finite sequence of data tuples, called *in-flight data*, which can be used for producing output results. Every graph has special operators that act as *sources* and *sinks* by subscribing to input streams or committing results externally.

An operator continuously ingests stream sequences, constructs finite views (i.e., windows) over them, applies an algebraic transformation, and outputs streams as a result. The operators can be *stateless* if they do not require state to compute their results, e.g., `PROJECTION` (π), `SELECTION` (σ); or *stateful* if they have to remember tuples from previous data to produce concise results and change their state for each event they process, such as `AGGREGATION` (α), `GROUP-BY` (γ), `JOIN` ($\bowtie$). Operators with explicitly defined windows assign the upper bound of the window as a timestamp to the tuples of the *derived streams*. Operators with unbounded

windows [21] (i.e., windows that contain all tuples from the beginning of a stream) assign the initial timestamp to the *derived streams*.

An important operator's characteristic is its selectivity because it can lead to optimizations such as executing specific operators earlier to reduce the intermediate results produced. Using selectivity, we measure the data tuples produced for every data tuple that arrives as an input. For example, operators such as `SELECTION` usually have selectivity less than one, and operators like the Cartesian product of two streams (e.g., `JOIN`) can have selectivity greater than one. Finally, stateful operators with windows are considered *pipeline breakers* [165], i.e., operators that require (partial) result materialization with windows before the next operators can process their output in the context of stream processing [234].

Depending on the use case, the *sources* of an operator graph can be heterogeneous and represent data with different formats (e.g., JSON or wire protocols such as Apache Thrift [18], and Protocol Buffers [179]). To enable the data collection and distribution to multiple machines with low overhead and fault-tolerance semantics, messaging systems can be used as a temporal buffer between sources and the operator graph. Popular message brokers and queuing systems are Apache ActiveMQ [14], RabbitMQ [182], ZeroMQ [233] (used by Apache Storm [211] as the default messaging technology to inter-worker communication) and publish-subscribe systems such as Apache Kafka [16].

Another characteristic of dataflow graphs is accessing previous data either to correlate them with the latest results or to guarantee correct results upon failure (see Section 2.5). Different solutions can be used for data storage for SPEs, ranging from key-value stores, such as BigTable [53], to time-series databases (e.g., Druid [227]). However, the procedure of storing these data should not be on the critical processing path. Finally, the output results of a streaming application are published to *sink*(s), to visualize or further analyze them (e.g., using Grafana [96]).

Window types and measures. While *stateless* operators can apply computation over infinite data steams, it is infeasible to maintain the entire stream history for some *stateful* operators. It is, thus, common to keep only the most recent view of data using *window operators*. These operators form a sequence of finite subsets of an input dataset for operators to apply their transformation on each subset (e.g., `JOIN` or `AGGREGATION`). We refer to the rules [169] for generating these subsets as the *window definition*. First, we must define a windowing attribute that establishes the order among tuples from a stream: event or processing timestamps (i.e., timestamps attached during execution by the system). Then, we must define the measurement unit, such as the number of tuples or time intervals, followed by the edge definition (i.e., when a window starts/ends), and its progression step (i.e., how a window changes its contents).

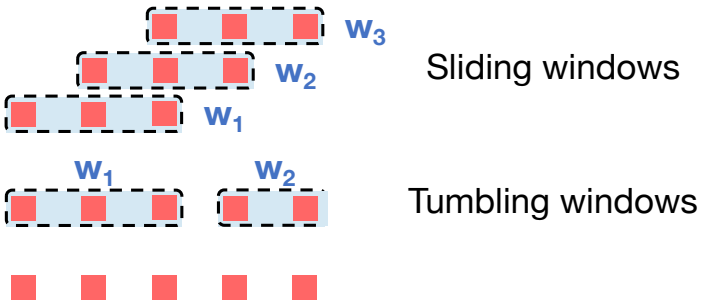


Fig. 2.2 Window types

In this work, we focus on two window types [7] as shown in Figure 2.2: *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. Our work supports only *deterministic* windows [46], i.e., windows that allow designating the beginning or end of a window upon tuple arrival.

When performing computation on windows, it is crucial to make progress (i.e., identify the start/end of a window) as quickly as possible since the significance of each individual tuple decays over time. Making progress affects the stateful operators that must wait until all data that belong to a window has arrived using concise synopses/summaries [19] before applying their transformation or emitting the output stream [30]. We next discuss the most general ways to make progress on streaming operators [7, 147, 169]:

- **Time-agnostic execution** is used when time is essentially irrelevant (e.g., all relevant logic is data-driven). It can be considered the simplest stream processing scenario, where timestamps are not required to compute the output.

- **Windowing by processing time** uses timestamps generated by the SPE during query execution. This type of processing simplifies the check of window completeness but leads to non-deterministic results across different runs of the same query [100].

- **Windowing by tuples** or *count-based* windows are a specific case of windowing by processing time, where tuples have monotonically increasing timestamps. In Figure 2.2, we show an example of a tumbling window of size three tuples and a sliding window of equal measure and slide one tuple.

- **Windowing by event time**, also referred to as *time-based windows*, uses the event timestamp $\varepsilon(t) \in \mathscr{E}$ from every tuple, which denotes when the event occurred. This

type poses many challenges when data arrives out-of-order as more guarantees are required for computing the results. In this thesis, we consider "in-order" or "slightly out-of-order" tuples that can be buffered and reordered before applying the operator's logic (i.e., deterministic window construction). Handling out-of-order data is beyond the scope of this work.

- **Punctuations** [215] (or low watermarks [6]) are introduced in a stream to define the end of substreams. They are assertions inserted to a stream that help the SPE identify whether tuples within a certain range of values can or cannot appear in the rest of the stream, thus giving the notion that a set of smaller finite streams constructs an infinite stream. In this dissertation, we use tuples similar to punctuation to perform fault-tolerance operations as discussed in Chapter 5 of this thesis.

- **Approximation algorithms** [30, 92] is the last type of processing that allows making progress with limited computing resources. Some approximation techniques are: sketches [83, 209], counting or sampling methods [77] and wavelets [90] (e.g., wavelet transforms over signals). Our work focuses on operators that produce accurate results.

Operator parallelism. There are three types of operator parallelism in a stream processing graph [107]: *pipeline*, *task* and *data* parallelism. Pipeline parallelism occurs when the execution of two tasks has interdependencies but can be overlapped if the first task pipes its output to the second. Task parallelism occurs when two or more non-pipelined operators execute concurrently. Finally, data parallelism occurs when multiple instances of an operator process different portions of the same data. In the latter case, an SPE needs operators with multiple output streams (e.g., `SPLIT/PARTITION`) and multiple input streams (e.g., `MERGE`).

Execution models. Any SPE can be classified into two distinct execution models: either the continuous dataflow model [80, 8, 211, 45, 49] or the micro-batch approach [217, 231, 137]. The execution model influences the core features of an SPE's architecture and defines the whole execution process, starting from when the computation is triggered (i.e., per tuple or per set of tuples basis).

With the continuous dataflow model, each operator is assigned to processing units (e.g., CPU cores or cluster nodes) as long-running tasks, also referred to as materialized tasks. These tasks are independent processing entities that can execute in parallel using stream partitioning and communicate using messages. The input data streams flow through a relatively static DAG of operators, keeping the mutable state locally and updating it accordingly. The event-based granularity of this execution model offers low processing latency because no communication with a centralized entity (driver) is required. In addition,

the one-tuple-at-a-time computation gives the user more expressiveness to define the desired operations over data.

The micro-batch approach replaces the tuple-at-a-time model with processing data in batches of input tuples. This model is influenced by the bulk processing framework of the MapReduce dataflow model [69] and became well-known through its implementation in Apache Spark [231]. The incoming data streams are discretized into batches with a predefined time interval that specifies the processing granularity of the tasks. These batches are then scheduled sequentially to the available processing resources by a centralized driver based on data locality [229], i.e., execute tasks where data resides. Similar to the continuous dataflow model, query execution can be described by a DAG of operators, which can be translated into multiple map-reduce phases or custom operators (e.g., join) chained one after the other. This model requires repeated communication with the centralized driver, so synchronization barriers are introduced between the computation stages, increasing processing latency. On the other hand, micro-batching provides native fault-tolerance, straggler elimination (i.e., by restarting slow tasks), high throughput, and the unification of batch processing and streams without much effort.

In this thesis, we adopt a variation of the micro-batch approach: each pipeline fragment (i.e., a subgraph of the dataflow DAG) maintains its own centralized driver that schedules tasks to a pool of processing entities (i.e., CPU worker threads). In the context of scale-up deployment, this execution model provides comparable end-to-end latency and higher throughput results than the continuous dataflow model. It is interesting to investigate how this approach can be extended for a distributed design as future work.

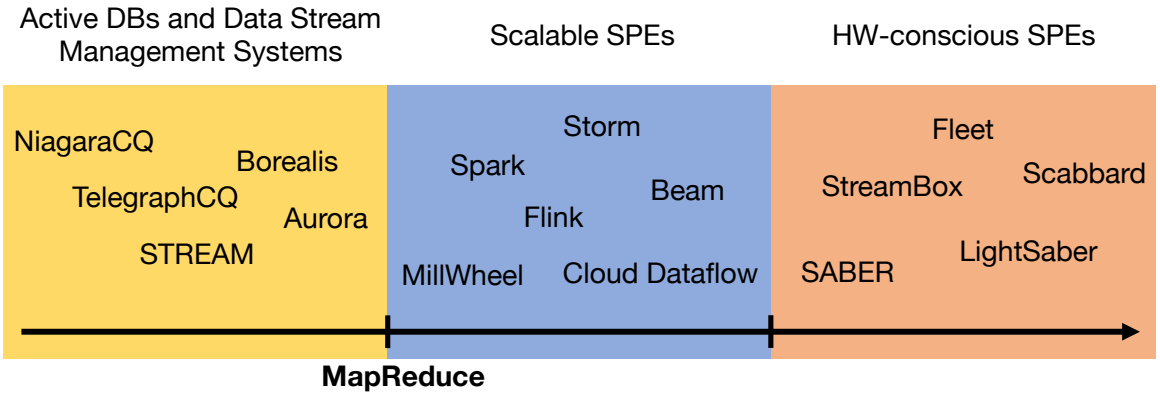


Fig. 2.3 Evolution of SPEs

2.1.3 The evolution of stream processing

This section provides a brief overview of how stream processing has evolved in recent years, becoming the fourth significant data-intensive application workload. This growth of real-time analytic use-cases, from card fraud detection to dynamic pricing, is translated into all cloud vendors' first-class support of stream processing technologies. While stream processing has been an active research field for more than two decades, only recently did we witness the emergence of hardware-conscious designs [137, 158, 236, 205, 208] that exploit parallel hardware, such as multi-core CPUs, GPUs, and FPGAs. Subsequently, we identify three distinct eras of SPEs, presented in Figure 2.3, and discuss their focus on different aspects of stream processing.

Since the notion of stream processing was introduced [203], databases and distributed systems have influenced the fundamental concepts behind SPEs greatly. The first generation of SPEs, also referred to as *Active Database Systems* (ADS) and *Data Stream Management Systems* (DSMS), emerged as extensions to traditional database management systems, enabling the computation of dynamic data (i.e., streams) using continuous operators. NiagaraCQ [56], TelegraphCQ [51, 150], and STREAM [22] have existed for decades but operated only on a single CPU core. These systems along with Aurora [2] and Borealis [224, 5], were research prototypes that set the architectural primitives for many modern SPEs while solving various challenges, such as window operations [126, 146], fault-tolerance [112], scale-up designs, and SQL extensions to represent a time-varying relational model (CQL [21]). This early research influenced the first commercial SPEs, such as IBM System S [39], Esper [76], Oracle CEP [167], and Microsoft StreamInsight [130, 9]. The latter commercial systems focused on Complex Event Processing (CEP) [65] and window queries on multi-core CPUs at the expense of weaker stream ordering guarantees for windows.

The second generation of SPEs was the result of joint research efforts from academia [80, 8, 231] and industry [163, 7, 6, 120, 49, 47, 211] that started after the introduction of the MapReduce dataflow model [69]. To accommodate growing data amounts of real-time analytics, distributed SPEs such as Flink [15] and Spark Streaming [231] scale out transparently hard-coded dataflow graphs to a cluster of nodes through appropriate data partitioning [45, 231]. The SPEs from this generation focus on distributed and fault-tolerant processing on shared-nothing commodity hardware while incorporating streaming and batch execution with a unified model [87, 7].

In the last years, we have witnessed a shift of SPEs towards hardware-conscious designs [238, 204, 137, 208, 205] to handle latency-critical applications and provide more predictable performance guarantees on single-node deployments [235, 166] with a lower resource footprint. With highly-parallel heterogeneous architectures and modern network

technologies (such as RDMA discussed in Section 2.2.3) becoming commonplace in data centers and cloud offerings [64], SPEs can exploit previously unseen levels of parallelism with terabytes of memory [216] and prevent scaling-out to multiple computing nodes. Consequently, researchers have proposed a range of parallel streaming algorithms [128, 94, 237] and SPE designs [137, 158, 236, 205, 157]. StreamBox [158] handles out-of-order event processing and BriskStream [236] utilizes a NUMA-aware execution plan optimization paradigm in multi-core CPUs. SABER [137] is a hybrid streaming engine that, in addition to CPUs, uses GPUs as accelerators. However, these systems implement ad-hoc aggregation and parallelization strategies, as we discuss in Section 2.4.1, yielding high performance only for specific workloads (e.g., tumbling windows). In addition, their design overlooks fault-tolerance as it is challenging to persist the streams and data required for recovery on a single node without affecting performance.

Therefore, designing a general-purpose relational SPE that transparently utilizes existing scale-up hardware to execute arbitrary streaming SQL queries with windowing [21] while providing correct results under failures is still an open challenge. In the remainder of this thesis, we focus on the last era of SPEs and study the design of a next-generation SPE that reduces system complexity and increases efficiency.

2.1.4 Stream applications summary

To sum up, in this section, we provide the basic concepts and terminology behind stream processing. We showcase that stream processing applications exhibit unique challenges compared to the classical "process-after-store" model due to their continuous nature. Based on the high throughput and low latency requirements, there is a demand for exploiting modern hardware to achieve such performance levels. Yet, developing a hardware-efficient and reliable scale-up SPE remains an open challenge, as existing solutions cannot provide a comprehensive solution with robust performance across different workloads. In the following (Section 2.2), we discuss the technology available in data centers for designing such an SPE.

2.2 Modern processors, storage and network stack

Let us now introduce the characteristics of modern hardware found in data centers today that are essential to understanding how to utilize their resources efficiently. First, in Section 2.2.1, we present the microarchitecture of modern processors. Then, we cover fundamental concepts behind the storage (Section 2.2.2) and network stack (Section 2.2.3) that will help us design an efficient scale-up SPE.

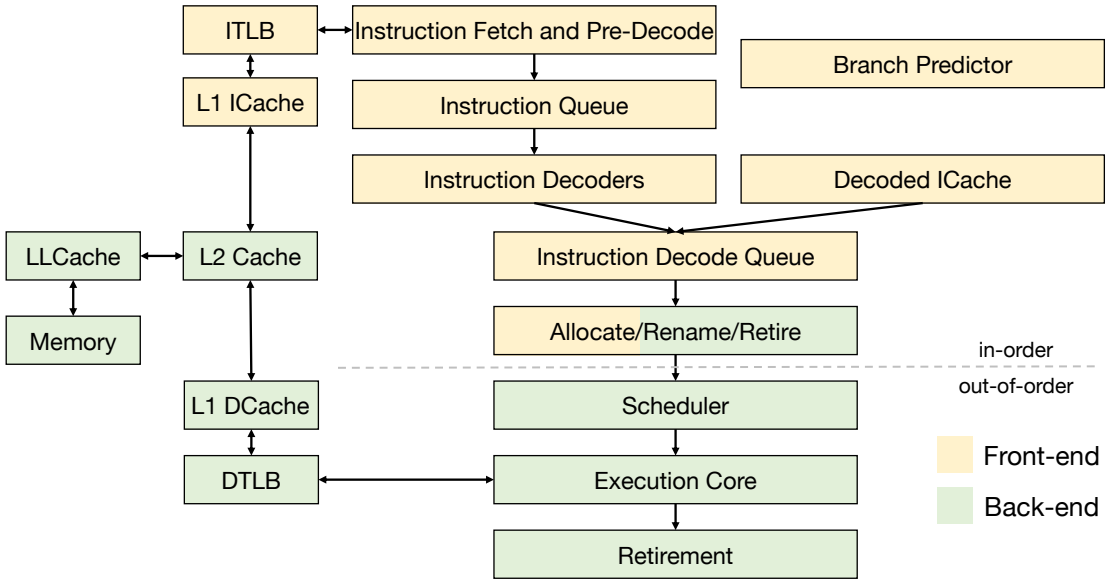


Fig. 2.4 CPU pipeline components

2.2.1 Hardware features of modern CPUs

In this section, we introduce the hardware components of modern CPUs, as shown in Figure 2.4. Even though the functionality of a program is unaffected by these components, their exploitation can accelerate program execution. Given that existing SPEs underutilize modern CPUs, as recent research revealed [234, 235], it is crucial to consider such hardware components when analyzing and designing stream processing algorithms and systems. Next, we discuss the hardware components exploited in this thesis, i.e., CPU microarchitecture, SIMD vector instructions, multi-core execution, and memory hierarchy.

CPU microarchitecture. Modern CPUs attempt to utilize their available hardware resources by employing pipelining: [2] (i) dividing the program instructions into a series of sequential low-level hardware steps or micro-ops (μops). These μops describe basic operations on CPU registers; and (ii) executing μops in parallel if they belong in different pipeline stages.

The pipeline stages include the following steps: (i) instruction fetch; (ii) instruction decode; (iii) instruction execution; and (iv) result writeback, i.e., write the result of an instruction back to CPU registers or memory. Figure 2.4 illustrates a simplified version of a high-performance CPU's pipeline, which is divided conceptually into the *front-end* and the *back-end* component [116].

The *front-end* pipeline is responsible for fetching and decoding the program code into μops. First, the instruction fetch units attempt to fetch instructions from the cache, i.e.,

[2] Also called instruction-level parallelism (ILP)

the instruction translation look-aside buffer (ITLB) and the L1 ICache, or memory. These instructions are pre-decoded and stored in the Instruction Queue. Finally, the instruction decoder units, i.e., instruction decoders and instruction decode queue, transform the pre-decoded instructions into μops. In more recent Intel CPU's microarchitectures, the *front-end* pipeline is accelerated using the Decoded ICache, which maintains up to 1.5K decoded μops without requiring the execution of the above steps. As the Decoded ICache is associated with instructions stored in L1 ICache, an L1 ICache miss invalidates the Decoded ICache.

The μops generated from the front-end component are fed into a process called *allocation*. After the *allocation* process, the *back-end* stages are responsible for monitoring when the execution units of a μop are available and execute them in an *out-of-order* manner. When the execution of a μop completes, its results are committed. This stage is also called *retirement*, and some operations may get canceled before retiring if there are *branch mispredictions*.

In the most recent Intel microarchitectures, applications can measure the hardware utilization of CPUs and derive which component stalls the CPU pipeline using dedicated *hardware counters*. The *front-end* can allocate up to four μops per cycle and feed them to the *back-end*. When there are stalls due to fetching and decoding operations, the execution becomes *front-end bound*, causing later stages to starve. If the *front-end* cannot deliver a μop because the *back-end* is not ready to handle it, the execution becomes *back-end bound*. There are different causes behind a *back-end stall*, such as stalls related to the memory subsystem (i.e., *memory-bound*) and stalls related to the execution units (i.e., *core-bound*). Another type of stalls is produced due to *bad speculation* from branch mispredictions, which results in operations failing to retire. Finally, applications can measure the number of retiring cycles that represent useful instructions using the hardware counters.

Data-level parallelism with SIMD instructions. Based on Flynn's classification on computer architectures [84], computer systems are divided into the following categories depending on the number of concurrent instruction streams and data streams: (i) a SISD system processes a single instruction stream on a single data stream; (ii) a SIMD system executes a single instruction stream on multiple data streams; (iii) a MISD system processes multiple instruction streams on a single data stream; and (iv) a MIMD system executes multiple instruction streams on multiple data streams.

Having discussed how modern CPUs perform instruction-level parallelism using pipelining, we now focus on the data-level parallelism achieved using single instruction, multiple data (SIMD) instructions. Most modern CPU architectures support SIMD instructions [116, 12] for a range of arithmetical, comparison, conversion, and logical instructions. The degree of the data-level parallelism is defined by the size of a SIMD register (i.e., the number of bits processed in parallel) and the data type size. For example, a 256-bit SIMD

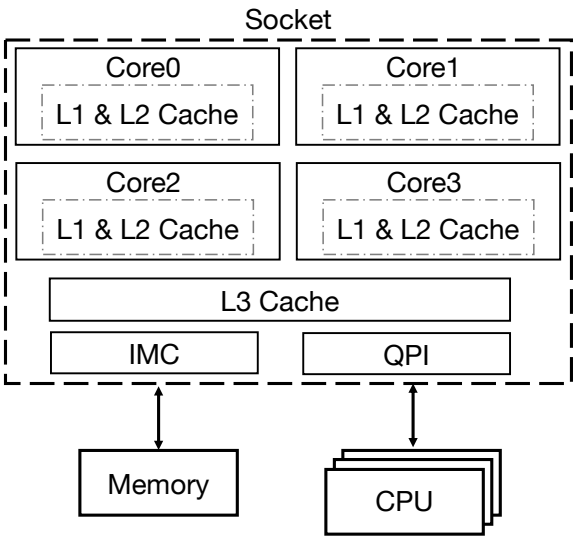


Fig. 2.5 CPU architecture

register executes a SIMD instruction on sixteen 16-bit or eight 32-bit data items. For the x86 architecture that we examine in this work, the Streaming SIMD Extensions (SSE) use 128-bit registers, the Advanced Vector Extensions (AVX) 256-bit registers, and the AVX-512 instructions increased the width to 512 bits. While SIMD instructions can improve performance in many scenarios, they also pose several challenges: (i) not all algorithms can be vectorized easily (e.g., due to data dependencies); (ii) in many cases, manual labor is required to vectorize algorithms as compilers do not generate SIMD instructions from typical programs; (iii) the developers must provide multiple implementations for different architectures; and (iv) SIMD instructions may incur higher latency compared to scalar execution.

Multi-core CPUs and memory hierarchy. Since the mid-2000s, scaling single-core clock frequency came to a halt [70, 190]. To combat these limitations, both multi-core and distributed architectures emerged. The first approach attempts to leverage the increased number of cores per chip, enabling thread-level parallelism as an alternative to frequency scaling. The second approach aggregates the resources of multiple machines with possibly similar hardware characteristics instead of scaling up hardware. Therefore, modern applications must utilize synchronization techniques to exploit multi-core or distributed parallelism. We next provide an example of multi-core CPU architectures.[3]

Figure 2.5 shows the architecture of an Intel processor [116] with four physical cores per socket. Each core has its own L1 Cache, L2 Cache, and execution units discussed above in the microarchitecture section while sharing a common L3 Cache with all other cores. The

[3]Distributed execution is orthogonal and complementary to our contributions toward improving the performance of modern SPEs.

Integrated Memory Controller (IMC) [116] is used to exchange data with the main memory with a bandwidth that depends on the memory frequency, the number of the bytes per width, and the number of the channels supported for the processor. Finally, the communication with other CPUs is established using the Quick Path Interconnect (QPI) interface [114] with up to up to 25.6 GB/s bandwidth.

The caches and memory mentioned above, along with persistent storage mediums, constitute a *memory hierarchy* with components that exhibit different response times. With this hierarchy, the lower the access latency, the smaller the available storage is. The CPU registers provide the fastest access and the smallest storage level, followed by the multi-level cache hierarchy, i.e., L1 DCache and L1 ICache (up to 64 KB), L2 Cache (up to 256 KB), and L3 Cache (up to 30 MB). If the data is not resident in the core's caches, it is fetched from the main memory with additional latency. When accessing the main memory, if multiple sockets are connected, their communication is based on the non-uniform-memory-access (NUMA) memory design. Each processor has its own local memory that is faster to access than non-local memory, which can be either memory local to another processor or memory shared between processors. Thus, the access latency is affected by the physical distance to the responsible memory controller. Finally, the highest latency is exhibited when fetching data from a disk. We conclude that this complex memory hierarchy requires a careful system design to achieve good performance.

2.2.2 Storage stack and asynchronous I/O

In the previous section, we discussed persistent storage mediums in the context of *memory hierarchy*. This section covers the fundamentals for disk persistence essential for designing an efficient storage layer. While most resources, such as input streams or storage devices, can be accessed over the network, SPEs must consider disk I/O to store and retrieve data streams and operator state efficiently. Next, we compare different approaches [133] to perform disk operations using: (i) `read()`/`write()` syscalls; (ii) `mmap()` syscall; (iii) direct I/O `read()`/`write()`; (iv) and asynchronous direct I/O.

Before analyzing these approaches, let us introduce the definition of a block device in Unix-like operating systems: a special file type that provides access to hardware devices, such as HDDs or SSDs, and some abstraction from their specifics [113]. While programmers can read or write a block of any size and alignment, a block device performs operations at the granularity of a sector (i.e., a fixed-size group of adjacent bytes, usually of size 512). For a file system, the smallest addressable unit is a block, consisting of multiple adjacent sectors with sizes ranging between 512 and 4 KB. A Unix-like operating system performs I/O using

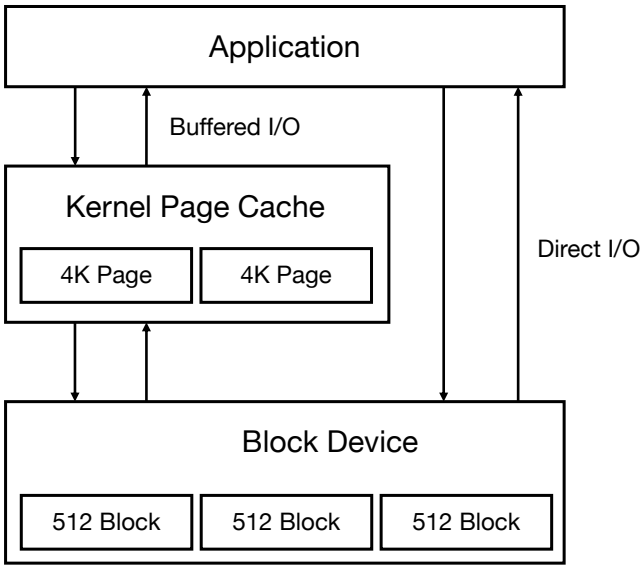


Fig. 2.6 Buffered and direct I/O

the virtual memory to map a program's memory addresses into physical addresses such as file system blocks using 4 KB pages. The virtual memory acts as a disk requests cache.

When using `read()`/`write()` syscalls, the kernel reads or writes to a file using a page cache, as shown in Figure 2.6. In the case of a read, if the data requested resides in the cache, the kernel immediately returns the result; otherwise, a page fault is triggered that blocks the calling thread until data is fetched from the block device. In the case of a write, the kernel copies the userspace buffers in the page cache and performs later the actual hardware write. The writeback of dirty pages can be forced using a `fsync()` syscall or by `close`ing the file, which is blocking the calling thread until completion.

Another approach to perform disk operations is to use `mmap()` syscall to map a section of the address space to refer to the contents of a file. If the requested data from the file is in the page cache, the kernel is bypassed; if a cache miss occurs, a page fault and thread blocking take place as above. Both approaches involve the kernel for caching and scheduling I/O operations and thus are called buffered I/O.

However, there are use cases (e.g., implementing a write-ahead-log [178]) that it is desirable to bypass the page cache by opening files with the `O_DIRECT` flag. This is called direct I/O and enables the disk controller to copy data directly to the userspace as demonstrated in Figure 2.6. By bypassing the kernel, an application avoids the extra copy of data in the page cache and reduces the CPU overhead. Using direct I/O requires that all operations are aligned to sector boundary, i.e., have a starting offset and buffer size of a multiple of 512. In addition, direct I/O can be performed asynchronously using Linux' non-blocking API

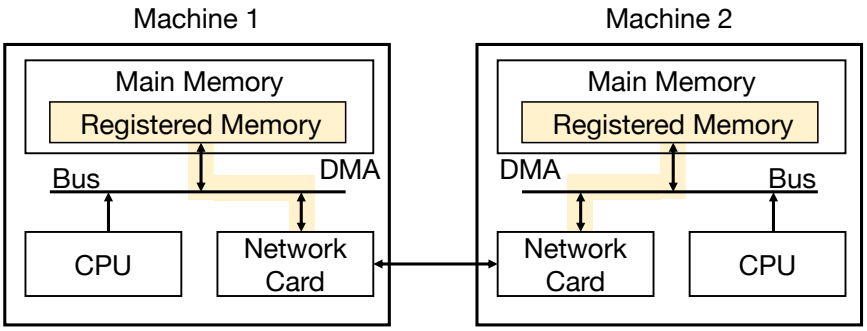


Fig. 2.7 Reading/writing data with Remote Direct Memory Access (RDMA)

with asynchronous notifications [132], which allows the pipelining of disk operations with computation without blocking the calling thread. In this work, we decide to use asynchronous direct I/O and perform the cache and memory management of disk operations manually to harvest the full potential of the hardware capabilities of modern SSDs.

2.2.3 Network stack

While an SPE can stream local files as its input, the most practical case is to use remote sources such as network sockets (e.g., POSIX sockets), distributed file systems (e.g., HDFS) or messaging systems (e.g., Kafka [16], Kinesis [11] or Pulsar [180]). Thus, it becomes crucial to understand how data streams are ingested over the network and accelerate this process to exploit the available hardware resources of a single node. This section aims to overview modern networks, explicitly focusing on hardware mechanisms such as Remote Direct Memory Access (RDMA).

With the emergence of modern network technologies, network I/O is no longer the bottleneck in many single-node scenarios [206, 157, 41]. RDMA-capable networks are becoming a commodity in data centers today [125] and provide 200 Gbps per-port bandwidth with microsecond latencies [26]. Such high-speed interconnects allow for fast stream ingestion, distributed operator implementations [36] and remote storage [135], with the potential to outperform main memory bandwidth [41]. However, existing cluster-based SPEs cannot saturate these fast interconnects [234].

As shown in Figure 2.7, RDMA enables direct memory access from the memory of a remote machine into that of another while bypassing both operating systems. In addition, RDMA supports zero-copy networking and can bypass the remote machine's CPU. This allows high throughput, low-latency networking without involving any work from CPUs, caches (which are not polluted), or context switches [35] and improves scalability, as a single node can be a passive participant for many connections [159]. Common RDMA implementa-

tions are Virtual Interface Architecture [63], RDMA over Converged Ethernet (RoCE) [25], InfiniBand [25], and iWARP [115]. Running such implementations over common network protocols such as TCP and UDP (e.g., via IPoIB) is possible but removes the aforementioned benefits because of system calls and data copies involved [187].

The efficient utilization of RDMA requires the use of the InfiniBand RDMA verb interface, which is a low-level abstraction for data transfer [73]. This interface provides: (i) one-sided communication using read, write, and atomic primitives; and (ii) two-sided communication with send and receive primitives. With one-sided communication, only the sender's CPU is involved, while both the sender and receiver participate with the two-sided communication. The application must register with the network card the main memory regions participating in the communication process to make them accessible for RDMA. These memory regions are pinned to prohibit swapping them out, and their address translation information is installed on the RNIC. The asynchronous nature of RDMA allows the pipelining of network communication with other operations such as computation, which results in high-performance data transfers. We adopt this asynchronous ingestion approach with two-sided communication in our system design.

2.2.4 Modern hardware summary

Today's network technologies enable data transfer with main memory bandwidth on scale-up servers. While the network is no longer the performance bottleneck in stream processing, existing designs fall short of exploiting modern hardware [234, 235]. Therefore, we identify a need for a fundamental redesign of SPEs. Given that the first step to hardware-conscious algorithms and systems requires a good understanding of the available hardware, we overviewed its characteristics. The following sections use the concepts introduced to perform profiling, determine a set of performance bottlenecks and mismatches found in modern SPEs, and motivate our solution for scalable and fault-tolerant single-node execution.

2.3 Window aggregation

After discussing the characteristics of modern hardware, we now focus on the execution of streaming applications and, more particularly, on windowed aggregation operators. Window aggregation [21], i.e., the calculation of running aggregates from a FIFO buffer [6, 192], is a core operator of analytical queries over continuous data streams (e.g., credit card transactions or click logs). The importance of this class of applications spans outside the field of stream processing to operations on persistent data such as time-series analysis [164, 202]. This

section covers the underlying concepts of window aggregation required for the remainder of this thesis. We provide background on the algebraic and mathematical properties of aggregate functions (Section 2.3.1), aggregation sharing (Section 2.3.2), prefix- and suffix-scans (Section 2.3.3), incremental computation (Section 2.3.4), and programming interfaces for implementing aggregate functions (Section 2.3.5). Finally, we discuss the limitations of existing solutions (Section 2.3.6).

2.3.1 Algebraic properties

An *aggregation function*, such as sum or max, is an algebraic operation that performs arbitrary computations over a finite set of input tuples S. The output of these computations is a summary of the tuples into a single aggregate result. Every aggregation function can be classified based on its algebraic properties [97, 202] into the following categories:

- a *distributive* aggregate function (e.g., sum) can be applied in a distributive manner: the result of applying the function on the set S is the same as the result produced from partitioning S in n (non-overlapping) subsets, applying the function on every partition to produce a partial result, and finally, on all partial results.
- *algebraic* aggregate functions can be computed from a number of distributive aggregations (e.g., avg from sum and count). Both distributive and algebraic functions have fixed-sized intermediate results.
- *holistic* aggregate functions have no constant bound on the storage size of intermediate results (e.g., rank or median). Even though handling such functions is beyond the scope of this work, we discuss in the following chapters how we can extend our contributions to incorporate *holistic* computations.

In this dissertation, we focus on distributive and algebraic functions, which can be further classified by their mathematical properties: (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*, $(x \oplus y) \ominus y = x$, $\forall\, x,y$. Distributive and algebraic functions are associative and based on these mathematical properties, we are able to perform incremental execution by reusing previous computed intermediate results, as we will discuss next.

2.3.2 Partial aggregation

Unlike stateless operators, window aggregation requires that all tuples in the window have arrived and, in some cases, are buffered before producing an output result. That property

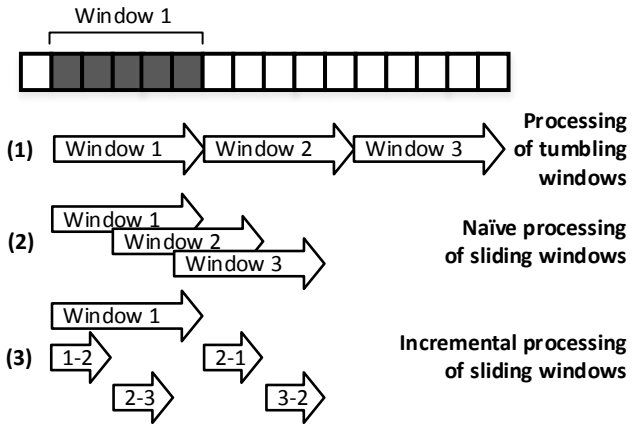


Fig. 2.8 The case for incremental window computation

makes it hard to parallelize the window aggregation operator itself. A common approach is to parallelize *on top* of the aggregation operator, i.e., process multiple windows in parallel. We show this process in Figure 2.8 using: (i) tumbling windows of size six tuples; and (ii) sliding windows with the same size and slide of two. When processing tumbling windows in case (1), Windows 1, 2, and 3 can be processed independently without redundant work. In case (2), the sliding windows are processed separately, but with redundant work: e.g., tuple 5 (i.e., the last tuple of Window 1) contributes to Windows 1, 2, and 3 and, thus, is processed three times. In the last case (3), the stream is broken down into fixed-size chunks, commonly referred to as *panes* [146] or partial aggregates, which are processed independently and in parallel.

The latter approach, i.e., case (3), is a typical execution strategy for window aggregation that exploits the associativity of aggregation functions: tuples can be (i) partitioned logically, (ii) pre-aggregated in parallel, and (iii) the per-partition aggregates, referred to *window fragment results* in this dissertation, can be merged. This technique is known as *hierarchical* or *partial* aggregation in relational databases [34, 74] and *window slicing* in stream processing. Several different partitioning techniques (also called slicing) have been proposed, e.g., Panes [146], Pairs [140], Cutty [46], and Scotty [212], 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 introduce the term *sashes* to refer to these *window fragment results* and use it throughout Chapter 4. In Figure 2.9, we present an example of *sashes* over a sliding window with six tuples window size and two tuples slide. The size of a *sash* is defined equal to that of the slide to increase the sharing opportunities between consequent window results. In the general case, multiple non-overlapping partial aggregates (i.e., panes) constitute a *sash* and adjacent *sashes* can be overlapping for sliding window semantics.

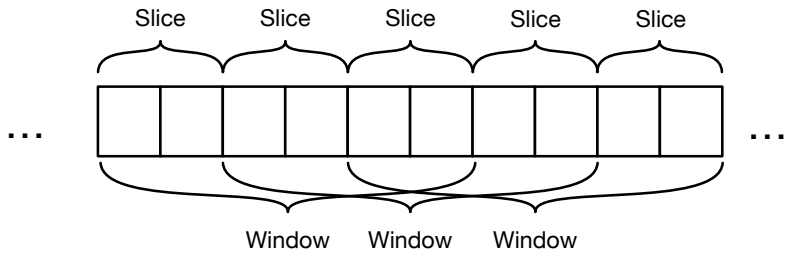


Fig. 2.9 Window sashes

2.3.3 Prefix- and suffix-scan

Before presenting the intuition behind incremental execution and existing algorithms, it is necessary to provide the definitions of the *prefix-* [42] and the *suffix-scan*.

Definition 2.3.1 (Prefix-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-scan* operations are defined as $PS[I_{\oplus}, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$.

Definition 2.3.2 (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 *suffix-scan* operations are defined as $SS[I_{\oplus}, a_{n-1}, (a_{n-1} \oplus a_{n-2}), ..., (a_{n-1} \oplus a_{n-2} \oplus ... \oplus a_0)]$.

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.

2.3.4 Incremental aggregation

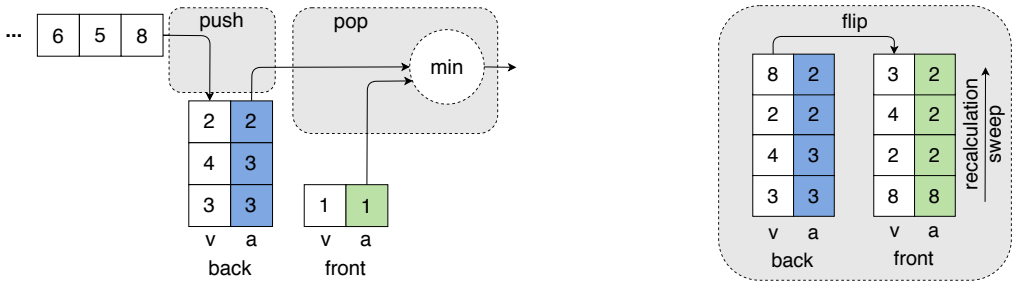
Although windows are finite subsets of tuples, they can be arbitrarily large. It is, therefore, preferable to use partial aggregation combined with *incremental processing* to reduce the number of operations (i.e., redundancy) required for window evaluation. *Incremental processing* involves maintaining and reusing intermediate results obtained by the unchanged parts of a window without re-evaluating them to increase the computational efficiency. Depending on the aggregation type, we can use different algorithms to incrementally summarize data in the granularity of tuples or partial results.

Table 2.1 provides an overview of different incremental aggregation algorithms: the second column denotes whether the algorithm uses different processing schemes for computing invertible and non-invertible aggregate functions; the third and fourth columns present the time and space complexities when computing either a single or multiple concurrent queries over the same data stream. In particular, Table 2.1 reports both the amortized and worst-case time complexities.

Table 2.1 Complexity of window aggregation approaches (n partial aggregates, q queries)

Algorithm	Queries	Time				Space	
		single		multiple		single	multiple
		amort.	worst	amort.	worst		
SoE [106]	inv.	2	2	q	q	n	qn
	non-inv.	n	n	qn	qn	n	qn
TwoStacks [106]		3	n	q	qn	$2n$	$2qn$
Slick-Deque [194]	inv.	2	2	$2q$	$2q$	n	$q+n$
	non-inv.	<2	n	q	qn	2 to $2n$	2 to $2n$
Slide-FlatFAT [202]	inv.	3	n	q	q	3n	$3n$
		$\log(n)$	$\log(n)$	$q\log(n)$	$q\log(n)$	$2n$	$2n$
Side [207]	non-inv.	3	n	q	qn	$2n$	$2n$

Single-query aggregation. For single-query invertible functions, Subtract-on-Evict (SoE) [106] reuses the previous window result to compute the next one by evicting expired tuples and merging in new additions. This approach has a constant cost per tuple for invertible functions but rescans the window if the functions are non-invertible ($O(N)$ complexity).



(a) Pushing/popping (b) Flipping

Fig. 2.10 TwoStacks algorithm

For single-query non-invertible functions, TwoStacks [4, 106] achieves $O(1)$ amortized complexity. As shown in Figure 2.10, 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 back stack's top. When the front stack is empty, the algorithm *flips* the back onto the front, reversing the values' order and recalculating the aggregates. The *flip* step requires re-scanning the contents of the whole window, but as it

occurs infrequently (i.e., when evicting expiring tuples from a window), it exhibits $O(1)$ amortized complexity.

Multi-query aggregation. While the previous algorithms process efficiently single query aggregations, when executing multiple queries over the same data stream, their time and space complexity is tightly coupled with the number of queries, as shown in Table 2.1. Therefore, another set of algorithms is designed to share the work between multiple queries. An example of such a multi-query application is a live-visualization dashboard that plots line charts of aggregates on time-series data at different zoom levels [212].

For multiple invertible functions, SlickDeque [194] 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 but exhibits large latency spikes due to the deque structure. FlatFAT [202] 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*- [42] and *suffix*-scan over the input, which hinders scalability with increasing window sizes. Finally, SlideSide [207] is our novel algorithm for multiple concurrent aggregate queries over the same data stream, which we discuss in Chapter 3. It uses a similar idea with FlatFAT but computes a running *prefix/suffix*-scan with $O(1)$ amortized complexity.

2.3.5 Programming interface for aggregation functions

Having discussed partial and incremental aggregation, next, we focus on the programming interface exposed by SPEs to express aggregation functions [202] and used throughout this dissertation. To implement an aggregation function, we decompose its operations into three functions: *lift*, *combine*, *lower*. If the function has the invertibility property, then an additional *invert* function is required to accelerate result sharing. We describe the interface using the average function as a running example:[4]

- *lift* computes a partial aggregate from an input tuple. When we want to compute the avg function over data values v, we perform the following operation $lift(v) = \langle v, 1 \rangle$.
- *combine* merges partial (i.e., intermediate) aggregates into another partial. For the avg function, if we combine two partial aggregates we get: $combine(\langle v_1, n_1 \rangle, \langle v_2, n_2 \rangle) = \langle v_1 + v_2, n_1 + n_2 \rangle$.
- *lower* turns a partial aggregate to the final output (e.g., $lower(\langle v, n \rangle) = \langle v/n \rangle$).

[4] Refer to [202] for an overview of how to implement different aggregation functions.

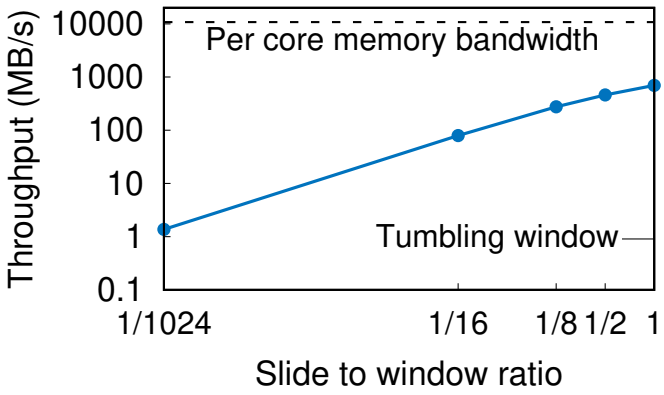


Fig. 2.11 Gap between memory and processing bandwidth for sliding windows

- *invert* removes incrementally a partial aggregate from another (e.g., $invert(\langle v_2, n_2\rangle, \langle v_1, n_1\rangle) = \langle v_2 - v_1, n_2 - n_1\rangle$).

2.3.6 Challenges in window aggregation performance

We now identify the limitations of existing window aggregation approaches for single and multiple concurrent queries, starting from the efficiency of single query execution. As discussed in Section 2.1.2, window aggregation partitions the input into finite subsets and calculates an aggregation result over them. While conceptually similar, tumbling and sliding windows are distinguished by their opportunities for intermediate result sharing as shown in Figure 2.8: in tumbling windows, every input tuple contributes to precisely one window aggregate; in sliding windows, a tuple contributes to more than one. In other words, the amount of necessary work per input tuple is bounded (upwards) by the maximum number of open windows that include a particular tuple. In the worst case, i.e., when the slide is one, this is equal to the window size. As a result, the performance of sliding window queries is typically dominated by factors other than the memory bandwidth, which distinguishes them from tumbling windows and classic relational aggregation optimization techniques.

To illustrate this mismatch, we implement a single-threaded running avg query[5] over a window of 1024 integers and vary the slide from 1 to 1024 (at which point the query turns into a tumbling window). Figure 2.11 shows that, for the tumbling window, the query saturates approximately a tenth of the per-core bandwidth. There is, thus, the potential to improve the performance of window aggregation by introducing data parallelism through SIMD instructions. As discussed in Section 2.2.1, SIMD-parallel implementations can increase the processing bandwidth of an algorithm. While tumbling windows are equivalent

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

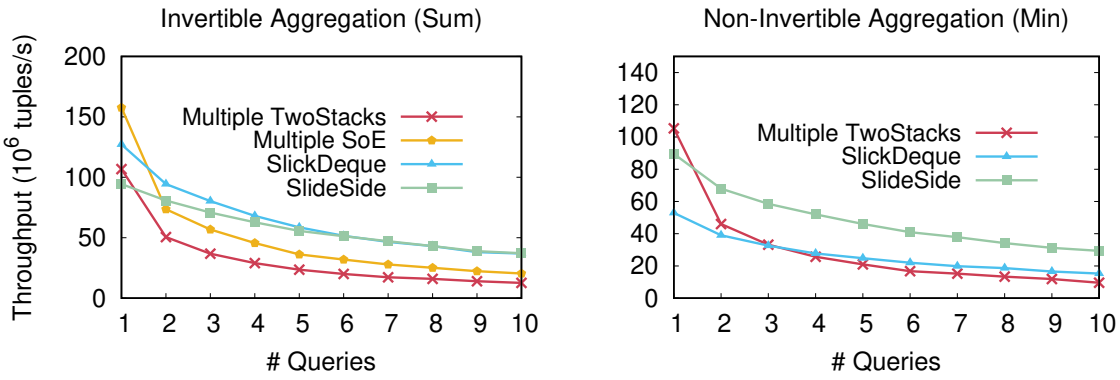


Fig. 2.12 Evaluating window aggregation queries

to relational aggregation and such optimizations are well documented [102, 177], there are no relevant approaches to achieve similar results for sliding windows. In particular, the gap between memory and processing bandwidth in the sliding window case is striking: an imbalance of more than three orders of magnitude.

Fortunately, there are many opportunities for intermediate result sharing in sliding window aggregation using the *incremental* algorithms and data structures [38, 202, 200, 106] discussed above. Considering our previous example of a running average query, an incremental algorithm does not recompute the average over 1024 values for each window slide. It instead adds the new values that entered the window and subtracts those that fell out of it. Incremental algorithms, however, commonly expose many control and data dependencies in the CPU instruction stream. These dependencies, in turn, hinder opportunities for efficient superscalar execution and SIMD parallelism, and make it challenging to perform CPU-efficient window aggregation, especially in the case of sliding windows or multi-query execution. Therefore, window aggregation can be executed either in parallel or incrementally, which exposes a trade-off between CPU- and work-efficient execution. However, given the demand from industry [40, 199] to increase CPU efficiency and prevent scale-out execution, it is crucial to resolve this trade-off.

The efficient window aggregation problem is amplified depending on the workload characteristics. Different aggregation function properties or window definitions (e.g., tumbling or sliding windows) make it challenging to design algorithms that adapt across workloads. As Table 2.1 indicates, there is no one size fits all solution: existing algorithms exhibit better performance or lower storage requirements in different scenarios. Given that current SPE designs [15, 211, 139, 24] pick a point in this trade-off space when evaluating window aggregation queries, they do not exhibit robust performance across query types. Figure 2.12 illustrates this problem by comparing the state-of-the-art approaches from above

for the evaluation of window aggregation queries [106, 194, 200, 4, 207]. The queries, taken from a sensor monitoring workload [122], calculate a rolling sum (invertible) and min (non-invertible) aggregation, with uniformly random window sizes of $[1, 128\text{K}]$ tuples and a worst-case slide of one. Some approaches exploit the invertibility property [106, 194] to increase performance; others [207, 200] efficiently share aggregates for non-invertible functions. To assess the impact of overlap between windows, we increase the number of concurrently executed queries.

As Figure 2.12 shows, each of the four presented 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. Based on our analysis, while the most efficient algorithms have a $O(1)$ complexity, they only achieve the best performance for specific window definitions. We conclude that there is a need to generalize window aggregation across different workloads.

2.3.7 Window aggregation summary

To sum up, while tumbling windows are amenable to "classic" relational query implementation techniques, the performance of sliding windows is more challenging to compute efficiently. Incremental algorithms introduce inherent control and data dependencies in the CPU instruction stream, as intermediate results from previous window instances have to be used to calculate the next ones. Thus, it is challenging to parallelize sliding windows in modern hardware with data-parallel approaches (i.e., SIMD intrinsics). In the case of multiple queries performing aggregation over the same stream, the optimization opportunities are obstructed even more, and accelerating such workloads has yet to be explored comprehensively [194]. Finally, existing techniques do not exhibit robust performance across different aggregation functions and concurrency levels. Thus, an SPE either performs poorly for various points within this design space or must maintain multiple algorithms using a complicated cost model to adapt to different workloads.

In Chapter 3, we describe how to achieve significantly better performance for state-of-the-art incremental window-aggregation algorithms through careful, hardware-conscious optimizations using HammerSlide. In the same chapter, we introduce a novel algorithm, called SlideSide, that extends TwoStacks for multiple concurrent queries over the same data stream. SlideSide accelerates intermediate result sharing by computing running *prefix/suffix*-scans over windows with $O(1)$ amortized complexity based on the algebraic properties of the aggregate functions. Finally, in Chapter 4, we introduce an abstraction that generalizes the

different state-of-the-art incremental approaches and generates code for efficient incremental execution by adapting to the workload characteristics.

2.4 Scalable data stream processing

In the previous section, we focused on window aggregation operations and the challenges behind their single-core execution. This section describes how SPEs scale complex applications with multiple streaming operators to many CPU cores or cluster nodes, focusing on parallel window aggregation. First, in Section 2.4.1, we present the parallelization strategies adopted by modern SPEs. We then discuss the limitations of existing approaches (Section 2.4.2 and Section 2.4.3), and summarize our findings in Section 2.4.4.

2.4.1 Parallelization strategies for stream processing

Let us now expand our analysis over the parallelization strategies employed to accelerate processing depending on the number of available processing units (i.e., workers). Every stream processing application can be translated into an operator graph (see Section 2.1.2), where the operators represent transformations over the input streams. Naturally, parallelizing *stateless* operators is straightforward: replicating the operator logic among a set of workers (i.e., *parallelism degree*), and assigning them tuples from the input streams based on scheduling policy (e.g., round-robin). On the other hand, parallelizing *stateful* operators is more challenging because of data dependencies between input, intermediate state, and output streams. The main problems that arise for stateful strategies are: (i) preserving the ordering semantics required by stream processing operators (i.e., precise windowing); (ii) expiring and updating windows concurrently; and (iii) balancing computations among workers regardless of the workload characteristics (i.e., number of distinct keys). In a distributed environment, these problems are amplified due to the lack of a global clock, leading to redundant operations [33] or approximate window semantics [231]. Distributed systems, such as Apache Storm [211] or Spark Streaming [231], lack clear definition and support for event-based window semantics and required extensions [213] or custom solutions.

The most common parallelization strategy for *stateful* operators is *partitioning-by-key* [24, 15, 211] . This approach requires a `PARTITION` operator that maps input tuples to workers based on a partitioning function (e.g., consistent hashing). The workers execute the same operations over distinct data partitions and create independent intermediate results called window fragments. The final step of this approach is to assemble the output stream by concatenating these intermediate results with a `MERGE` operator (e.g., in parallel using a

Table 2.2 Parallel window aggregation in stream processing systems

System	Shared memory	Single-node aggregation	Parallelization	Incremental	Slicing/query sharing
Flink [15], *Spark* [24]	✗	✗	partition-by-key	within window	✗
Cutty [46], *Scotty* [212]	✗	✓	partition-by-key	within/across window	✓
Borealis [33]	✗	✗	partition-by-pane	within/across window	✓
StreamBox [158], *BriskStream* [236]	✓	✓	partition-by-key	within window	✗
SABER [137]	✓	✓	late merging [234, 143], single-threaded merge	within/across window	✗
LIGHTSABER	✓	✓	late merging [234, 143], **parallel merge**	within/across window	✓

distributed file system). Other partitioning techniques involve partitioning by either *windows* or *panes* [33, 68] and attempt to load balance execution at the expense of redundant data sent between multiple nodes in the event of overlapping windows.

The second approach for parallel *stateful* operator execution is called *Late Merging* [234] and was first introduced in main-memory databases [143, 168] and then in stream processing [137]. In the first stage of *Late Merging*, the system partitions the input streams, and independent workers pull the produced partitions on their own to balance the processing load. Finally, the intermediate results from each worker are merged to construct the output stream. Compared to *partition-by-key* concatenation, the merging step of *Late Merging* is more challenging to parallelize because of the intermediate results' data dependencies.

2.4.2 Trading-off parallelization for incremental execution

Having introduced existing parallelization strategies employed by modern SPEs, we now focus on parallel window aggregation due to its importance in complex streaming analytics. As discussed in Section 2.3.6, window aggregation, by its very nature, can be executed either in parallel, if there is independent work, or incrementally, which introduces dependent work—but not both. Consequently, the work-sharing benefits between overlapping windows must be traded off against the parallelization benefits. This section extends the notion of parallelism from SIMD-parallel to multi-core execution by summarizing the design decisions of existing SPEs for parallel window aggregation in Table 2.2.

We first present Table 2.2: the second column denotes whether a system considers shared-memory architectures; the third column describes whether data must be available on a single node for aggregation; 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 [24] and Flink [15]) distribute processing to a shared-nothing cluster and parallelize it with key-partitioning. This approach requires a physical partitioning step between operators that enables both intra- and inter-node parallelism. However, with key-partitioning, 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 directly aggregate individual tuples into full windows, following the bucket-per-window approach [147, 148]. This aggregation step becomes expensive for sliding windows with a small slide when a single input tuple contributes to multiple windows.

Slicing frameworks, like Cutty [46] and Scotty [212] developed on top of Flink, reduce the aggregation steps for sliding windows and enable efficient inter-query sharing. Eager slicing [46] performs incremental computation both within and across window instances using FlatFAT; lazy slicing [212] evaluates only the partial aggregates incrementally and yields better throughput at the cost of higher latency. However, the parallelization method is still key-partitioning, and these approaches perform aggregation on a single node. Therefore, they suffer from the overheads of distributed execution without the ability to scale out to multiple nodes.

Borealis [33] applies slicing differently: it extends *panes* [146] for distributed execution. However, *partitioning-by-panes* still requires a centralized partitioning step that limits performance, introduces additional data transfers for overlapping windows, and couples window semantics with performance.

Scale-up systems, such as StreamBox [158] and BriskStream [236], are designed for NUMA-aware processing on multi-core machines. Both systems use key-partitioning for parallelization and use the bucket-per-window approach for incremental computation, overlooking the optimization opportunities of window aggregation.

SABER [137] 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 (i.e., *late merging* step).

SABER's approach 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 [33, 146]. SABER decomposes the operator functions into: (i) a *fragment function*, which processes a sequence of window fragments and produces fragment results (or *sashes*); and (ii) an *assembly function* (late merging [234, 143]), which constructs complete window results from the fragments and reorders them based on window semantics. 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 systems apply key-partitioning for parallelization, this approach does not exploit all the available parallel hardware. Parallelizing by panes in a distributed environment introduces additional data transfers that hinder performance. In addition, when evaluating overlapping windows, no system from Table 2.2 combines effectively partial aggregation with incremental computation, which results in sub-optimal performance, as we show in Chapter 4.

2.4.3 Revisiting the COST of parallel execution in modern SPEs

Let us now drill down on the performance characteristics of existing SPEs to understand their bottlenecks when parallelizing stream processing operators on modern CPUs. We compare the performance of efficient scale-up SPEs, such as SABER and StreamBox, with that achieved by popular distributed SPEs, such as Apache Flink [15] or Apache Spark [17], in terms of parallel (i.e., multi-core and distributed) and single-core execution.

Hardware. We perform the experiments on six servers (one for coordination and five for execution), each with two Intel Xeon E5-2660 v3 2.60 GHz CPUs with 20 physical cores, a 20 MB LLC cache, and 32 GB of memory. The machines are connected with 10 Gbps Ethernet. We used only eight cores per node for the distributed experiments, as we did not see any significant change in throughput after this number.

Workload. We evaluate Yahoo Streaming Benchmark (YSB – see Section A.1) [58] with SABER (without its GPU support) against Apache Spark (version 2.4.0), Apache Flink (version 1.3.2), and the latest version of StreamBox. YSB emulates an advertisement application with four operators: `SELECTION`, `PROJECTION`, `JOIN` (with relational data) and `AGGREGATION` (a windowed count). In our implementation, input tuples have 128B size.

We designed our experiments with the explicit intention of isolating the performance of SPEs from external influences. For that purpose, we conduct the experiments as follows:

- For SABER, we initially generated the data on a separate machine. Since only a single CPU core manages to saturate the 10 Gbps network connection (8.3 million tuples/s), we instead generate data in-memory.

- For Spark and Flink, we follow the approach from previous blogposts [99, 228]. More specifically, for both systems, we use a static table for the join instead of Redis and generate data in memory. Finally, we enable object reuse for Flink.
- For StreamBox, we extend the word count example and create our source and operators. StreamBox represents windowed operations as session windows with watermarks that define the end of a window (see Section 2.1.2). To avoid a bottleneck, we generate ordered data with at least one source thread per worker. We emit watermarks every 10 seconds to create 10-second tumbling windows.

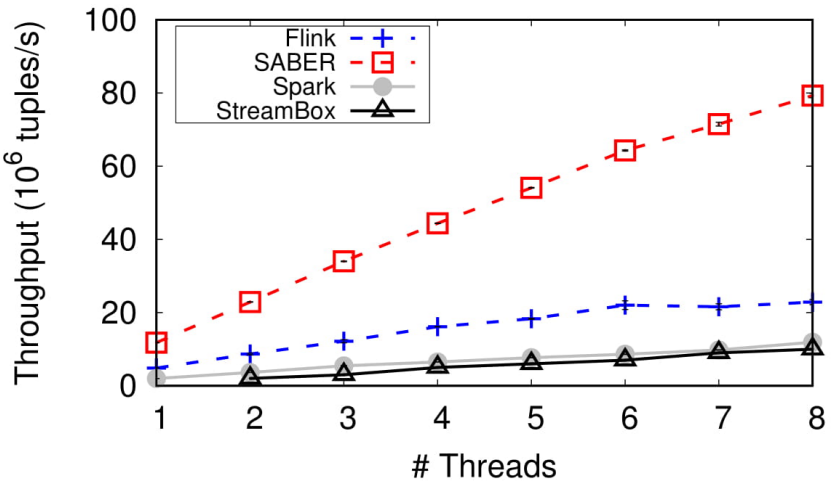


Fig. 2.13 Single server throughput for YSB

Throughput Comparison. Figures 2.13 and 2.14 show the scalability of the four systems in terms of throughput as we increase the number of cores. With a single node, Flink performs better than both Spark and StreamBox (around 12 million tuples/s), increasing throughput by more than 1.9×. Compared to the other systems, SABER exhibits nearly 7×, 3×, and 7× better throughput than Spark, Flink, and StreamBox, respectively, as it processes almost 79 million tuples/s. SABER surpasses the best single-node throughput of the other systems with just two CPU cores and competes for the performance of distributed execution.

Apart from not exploiting the memory hierarchy and minimizing data copying, the throughput of both Flink and Spark is affected adversely by communication and serialization overheads. These overheads are expected -– their distributed designs try to take advantage of the aggregated performance of multiple nodes. On the other hand, SABER binds worker and generator threads to CPU cores to minimize the memory accesses beyond the L2 cache. In addition, it maintains an input buffer of 512 KB, which ensures that data is kept

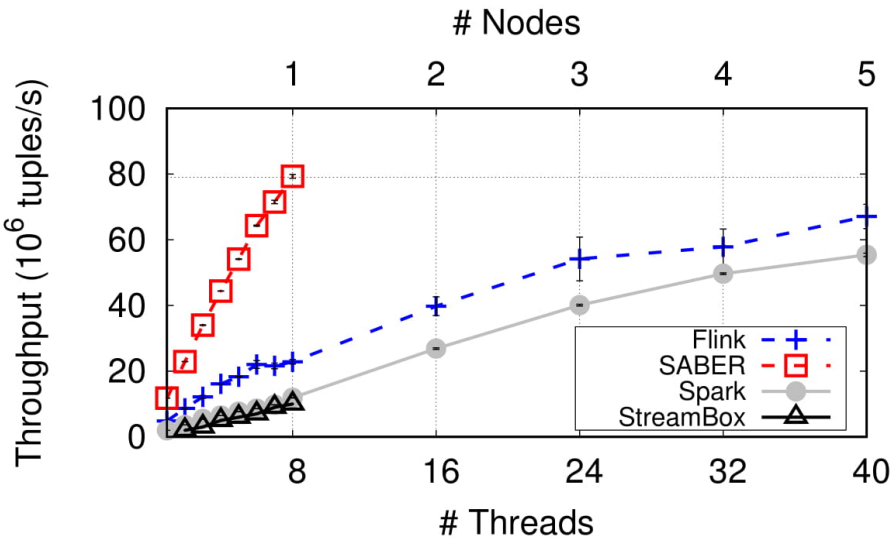


Fig. 2.14 Cluster throughput for YSB

in the LLC. SABER uses atomic operations to write and read parts of this buffer with negligible synchronization cost. Its design showcases the benefits of single-node scale-up SPEs. However, we still want to investigate missed optimization opportunities for such a design with the following experiment.

Table 2.3 Single CPU core throughput for Yahoo Streaming Benchmark

	Spark	*Flink*	*SABER*	*Handwritten C++*
Throughput (10^6 tuples/s)	2	4.8	11.8	39

Single-core performance. According to the previously-proposed Configuration that Outperforms a Single Thread (COST) metric [155], we analyze the performance of these systems by comparing their single-core implementations with a handwritten C++ program. The C++ implementation processes 39 million tuples/s (i.e., it is 4× faster than SABER) on our testbed server. Table 2.3 compares our reported throughput results with that of the single-threaded C++ implementation: there remains a large performance gap between the handwritten code and current SPEs. As we see from these results, removing expensive function calls, instruction cache misses (due to JVM), and memory cache misses (caused by serialization, copying, and object allocation) already causes a major performance benefit. We confirm that existing SPEs underutilize cache hierarchies, especially when parallelizing window operations. Consequently, we need to revisit their design in multi-core architectures. In Chapter 4, we perform a more extensive analysis to identify the bottleneck sources and provide a more robust scale-up design.

2.4.4 Scalable data stream processing summary

This section reviewed existing parallelization strategies of modern SPEs and focused on parallel window aggregation. More specifically, we presented that SPEs implement ad-hoc strategies for window aggregation. As a result, they only achieve high performance for specific queries depending on the window definition and the type of aggregation function. We identified, thus, the importance of designing a system that generalizes the design space of existing aggregation strategies. In addition, from our experiments in Section 2.4.3, we demonstrated a performance gap between existing SPEs' implementations and handwritten code, which is yet to be analyzed if we want to design hardware-conscious SPEs. To address these challenges, in Chapter 4, we design compilation-based techniques to keep data in CPU registers as long as possible, maximizing data and code locality [165], and balance parallelism with incremental processing on modern scale-up architectures.

2.5 Fault-tolerant stream processing

In the previous sections, we discussed the emergence of hardware-conscious scale-up designs for single-node SPEs [137, 158, 236, 205, 157] (see Section 2.1.3) that rival the performance of cluster-based deployments (see Section 2.4.3). The performance gap becomes more evident when performing stream ingestion or remote storage [135], as existing cluster-based SPEs cannot saturate high-speed networking [234], such as RDMA [125, 36]. In contrast, single-node SPEs yield up to an order of magnitude higher performance with fewer resources and lower maintenance costs [175]. Such high execution efficiency is achieved by avoiding abstractions for distributed processing and incorporating techniques such as just-in-time (JIT) code generation [205, 100], as we present in Chapter 4.

Despite these advantages, single-node SPEs have seen limited adoption in practice due to a lack of fault-tolerance mechanisms that guarantee correct results upon system failures [197, 15, 118]: *repeatable results* and *high availability* are critical requirements of real-world streaming applications, as discussed in Section 2.1.1. Given the long-running nature of stream queries, the failure of computing nodes is common. Existing cluster-based SPEs achieve at-least-once or exactly-once delivery semantics using different resiliency strategies [112, 48] by persisting input tuples along with the computational state [45, 80] or logic [230] (i.e., state checkpointing). Systems typically offload persistence to: (i) external distributed messaging systems for streams (e.g., Kafka [16], Kinesis [11] or Pulsar [180]); and (ii) external stores for operator state (e.g., RocksDB [78], Faster [50], or BigTable [53]). The use of external systems for persistence introduces overheads [185, 183, 72] that increase the size of scaled-out deployments. While the same persistence approaches could be used

for single-node SPEs, relying on an external cluster-optimized system for persistence, such as Kafka, counteracts the benefits of a single-node deployment and misses optimization opportunities, as we discuss below.

This section presents the fault-tolerance approaches for stream processing and explores the challenges faced when designing a single-node fault-tolerant system. First, we describe our failure model (Section 2.5.1). We then discuss how fault-tolerance is realized in SPEs and present the limitations of existing solutions (Section 2.5.2).

2.5.1 Failure model and processing guarantees

SPEs [45, 231, 80, 137, 118] execute continuous queries that translate into the *operator graphs* that we introduced in Section 2.1.2. We now define the operator graph used by our failure model.

Definition 2.5.1 (**operator graph**). An operator graph is represented as a graph $q = (O, S, B)$, where O is a set of *operators*, S is a set of *streams* and B is a set of *feedback channels* used for acknowledgments between operators. The graph's nodes $o \in O$ represent computations over data streams. The directed edges between the nodes (i.e., streams $s \in S$ and feedback edges $b \in B$) are reliable FIFO communication channels.

The operators of such a graph can be stateless (e.g., `SELECTION`) or stateful (e.g., `AGGREGATION`) and maintain arbitrary state, usually defined with finite windows [21] of tuples. However, given typical failure rates in large data centers [91], stateful operators pose a challenge for providing correct results under failures.

We consider non-deterministic software or hardware failures [134] that cause an SPE node to *fail-stop* [75], such as exhausting a node's memory or a node crashing. We assume each SPE node is connected with external sources and sinks via a high-bandwidth, low-latency network and has access to stable storage that survives failures (i.e., remote flash storage [101, 10, 138, 93]). The network layer provides a reliable FIFO delivery protocol (e.g., TCP/IP protocol).

Upon failure, operators must resume processing from the point they failed. For stateful operators, recovery requires redundant storage [210] of: (i) the computational logic to replay past tuples; and (ii) the computational state [118] to avoid data replay, as the state can depend on the entire stream history.

SPEs can achieve high availability [112] using *passive standby*, *active standby*, or *upstream backup*: with passive standby, streams (and state) are maintained in stable storage or the memory of another node; with active standby, redundant nodes are deployed that

receive and process the same streams as the primary ones; with upstream backup, each node retains its output and, in case of failure, restores the downstream node's state by replaying it.

When designing a fault-tolerant SPE, it is crucial to consider the processing guarantees it can support. There are three different types of recovery guarantees: (i) *at-most-once* (or *gap recovery*), if there is no guarantee that all the tuples will be delivered and eventually processed; (ii) *at-least-once* (or *rollback*) when all tuples are guaranteed to be processed without handling duplicate results; and (iii) *exactly-once* (or *precise recovery*) when the system guarantees that all tuples are reflected exactly once on the state and final output.

To fully mask the effects of failure, an SPE must remove duplicate tuples when restoring the state. Fragkoulis et al. [85] distinguish between *exactly-once state* and *output*: with the former, the system restores its state consistently but cannot handle duplicates in the output, whereas in the latter case, it avoids duplicates. Providing *exactly-once output* is also referred to as the output commit problem [75], *precise recovery* [112] or strong productions [6]. In the following sections, we assume *exactly-once output* processing guarantees.

2.5.2 Failure recovery in SPEs and their limitations

We now examine how SPEs achieve fault-tolerance with exactly-once results and discuss the challenges for single-node designs. For data replay upon failures, modern SPEs utilize external distributed messaging systems (e.g., Kafka [16]), coupling, thus, their performance to how fast such systems can ingest and transfer data. To provide exactly-once output, SPEs outsource the problem to external sinks that support transactional or idempotent writes [139]. Finally, for exactly-once state, the four most common fault-tolerance approaches are:

(i) Transaction-based: Trident [213, 211] and MillWheel [6] remove duplicates by assigning unique identifiers to tuples and committing state updates or produced tuples to an external transactional store [53]. Both approaches, however, add non-negligible overhead with large state and increase end-to-end latency.

(ii) Lineage-based: Spark Streaming [24], StreamScope [149], and TimeStream [181] track and persist the input/output dependencies of operators (i.e., lineage [230]) before execution, which would compromise performance for scale-up designs. Using the lineage of the failed operators, these systems restore the previously computed state by re-executing tasks.

(iii) Checkpointing: Flink [44] and IBM Streams [120] use a distributed protocol for global checkpointing [52] that asynchronously persists operator state with epochs, referred to as *aligned checkpoints*, on distributed file systems (e.g., HDFS). Other approaches [80, 81, 163] log tuples from streams for better runtime performance at the expense of higher recovery

times [85], called *unaligned checkpoints*. With an embedded key-value store, such as RocksDB [78], the previous systems can checkpoint their state incrementally [210].

(iv) Changelog-based: To enable state recomputation without persisting state dependencies, Kafka Streams [118] persists state metadata in a changelog, which is stored in Kafka [139]. Although its design combines computation with storage, the use of Kafka as the messaging system between operators increases end-to-end latency.

The above approaches require external cluster-optimized systems for persistence. Therefore, relying on them in a scale-up design not only counteracts the benefits of a single-node deployment but, in some scenarios, can limit performance. For example, in the case of stream ingestion, we observe that a single Kafka node cannot support the performance requirements of modern single-node SPEs (as discussed in Chapter 5). While it is possible to scale out the Kafka deployment to increase its throughput linearly through stream partitioning, this requires a large cluster (with associated maintenance costs) just to support a single SPE node.

A strawman solution is to design a "self-contained" fault-tolerance mechanism for a single-node SPE in which the engine persists all input data streams (and temporary processing state) to stable storage to recover processing after failure. We observe that, for such an approach, disk I/O bandwidth becomes the limiting factor for many queries, capping performance (e.g., to 950 MB/s for the experiments shown in Chapter 5). While I/O bandwidth can be increased through hardware solutions (e.g., NVMe SSDs [226] or RAID [170]), this also increases the maintenance costs.

Thus, designing a fault-tolerant single-node SPE with exactly-once semantics while retaining this performance is an open challenge, especially when considering the limited I/O bandwidth of a single node. In addition, such a system is expected to provide sub-second recovery upon failure. Lowering the system's downtime is crucial given the time-sensitive nature of real-time workloads that must report results with low end-to-end latency, such as online gaming [48].

2.5.3 Fault-tolerant stream processing summary

To conclude this section, while existing fault-tolerance approaches offer strong guarantees under failures in a distributed environment, they face limitations for single-node SPEs. First, they rely on external messaging systems [139, 180] to create fault-tolerant sources. These messaging systems require non-trivial tuning [71, 37] and do not maintain compact representations of stream data, which can lead to higher recovery times [152]. Second, they use key-value stores for state management, often not designed for stream applications [124]. This limits performance [99, 151, 127] and misses optimization opportunities.

As existing approaches cannot guarantee the performance requirements of hardware-conscious single-node SPEs, in Chapter 5, we propose SCABBARD, a new single-node fault-tolerant SPE that provides exactly-once semantics without compromising processing throughput. SCABBARD's fault-tolerance approach is to persist input streams and transient operator state to an SSD using novel persistence abstractions and adaptive compression techniques that reduce disk bandwidth.

2.6 Summary

In this chapter, we described the basic concepts related to stream processing, focusing on the trend to design hardware-efficient scale-up systems. We then introduced the hardware features of modern CPUs, storage, and network technologies, which are essential for developing the next-generation SPEs. Next, we discussed the fundamentals behind window aggregation and existing approaches while presenting the challenges of single-core execution. We then analyzed existing parallelization strategies and their limitations on multi-core architectures. Finally, we traced existing solutions for fault-tolerant stream processing and described the different approaches used by SPEs to achieve exactly-once results upon failures.

Using a range of real-world and synthetic data and applications, we experimentally demonstrated the limitations of existing solutions. We described how state-of-the-art aggregation approaches leave substantial room for improvement, starting from the single-core execution. Experimenting with an advertisement streaming application [58], we revealed existing designs' limitations for parallelizing window computations (both at a data and task level). Based on our observations, we identified the challenges addressed in the next chapters.

Chapter 3

Efficient Window Aggregation and Multi-Query Sharing

In the previous chapter, we discussed the challenges of single-core window aggregation [21] and the missed optimization opportunities of existing solutions. Window aggregation refers to the calculation of running aggregates over FIFO data streams (i.e., in-order) [6, 192] using windows. This class of operators is one of the most common in complex streaming applications. Therefore, given the ever-growing amount of data collected and analyzed in real-time [186] and the bottlenecks that emerge from performing window aggregation [212, 40, 199], it is crucial to ensure scalability without sacrificing the response latency.

This chapter revisits the design of window aggregation operators on modern CPUs: we examine hardware-conscious stream processing in the context of CPU- and work-efficient single-core execution. CPU-efficient execution refers to exploiting the hardware components from Section 2.2.1, such as data-parallelism. Work efficiency refers to minimizing redundant operations, such as computations upon overlapping windows, using incremental execution. In particular, we target two distinct workloads: single and multi-query applications. An example of a single-query application is the calculation of statistics over user click logs [95, 49, 6] or compute cluster traces [220] using a sliding window. Regarding multi-query applications, users want to perform the same aggregate functions over a data stream with different window definitions. Typical multi-query applications debug a stream, report near real-time behavior over small and large windows (e.g., a second vs a day), or support live visualization dashboards for different periods. For the latter use case, a dashboard can report analytics on time-series data at different zoom levels [212] or various telemetries from the same IoT device reading [29].

As discussed in Section 2.1.2, data streams are conceptually infinite, and thus, partitioned into finite subsets of tuples, called *windows*. A window has a *definition*, which maps each

input tuple to a window *instance* that yields a result upon aggregation. Windows can be distinguished by whether their instances are disjoint ("tumbling windows") or not ("sliding windows"). Tumbling (a.k.a. fixed) windows slice up the input stream into segments with a fixed size temporal length (static *window size*). Sliding (a.k.a. hopping) windows generalize tumbling windows by specifying a *slide* parameter in addition to the size that specifies the distance between the start of two windows.

Unlike tumbling windows that are amenable to classic "relational" query optimizations, sliding windows introduce repeated computations that hinder optimizations and create a tension between data parallelism and work efficiency. While employing an incremental algorithm to prevent redundancy is possible (see Section 2.3.4), such algorithms lead to control and data dependencies in the CPU instruction stream that make it challenging to introduce data-parallelism (e.g., SIMD intrinsics). Therefore, incremental execution reduces latency but affects CPU efficiency negatively, compromising throughput performance. We observe the consequence of this trade-off in Figure 2.11: there is an orders of magnitude gap between memory and processing bandwidth.

As a step further, evaluating multiple concurrent queries amplifies the complexity of window aggregation, introducing additional data dependencies that harm query performance. As we identified in Section 2.3.4, sharing efficiently intermediate computations over concurrent queries has not been explored comprehensively. There is, thus, a mismatch between continuous applications that aggregate data and the available techniques of performing efficient window aggregation for overlapping windows in industry [40, 199]. This results in expensive scale-out approaches and increased end-to-end latency.

Opportunity. In this chapter, we argue that there is a need for efficient single-core execution of window aggregation. By developing hardware-conscious window aggregation operators, SPEs can minimize their resource footprint, prevent scaling out to multiple computing nodes, and reduce the end-to-end latency of time-critical applications. Therefore, we identify as the first step towards hardware-efficient SPEs the design of such operators and target the following mismatches: (i) CPU-efficient execution; and (ii) operation redundancy.

Requirements. To sufficiently support window aggregation for both single and multi-query workloads, our approaches must address the following requirements (**R1**-**R3**):

1. Minimize end-to-end latency and maximize throughput regardless of the workload characteristics, i.e., aggregation function type or the number of concurrent queries.

2. Ensure both *work* and *CPU efficiency* without sacrificing general applicability (i.e., support all *associative* aggregate functions).

3. Increase *sharing opportunities* and prevent *repeated computations*, even when processing multiple concurrent queries. An incremental algorithm should determine the order of evaluating different window results to maximize work efficiency.

The remainder of this chapter is organized as follows: in Section 3.1 we provide an overview of our proposed approaches. We then conduct a performance analysis of state-of-the-art incremental algorithms in Section 3.2. Based on our findings, we develop and present HammerSlide in Section 3.3 that addresses the **R2** requirement. Section 3.4 introduces SlideSide, our novel multi-query incremental algorithm (**R3**). We evaluate our approaches and present advice on how to select the most appropriate aggregation algorithm in Section 3.5. The chapter finishes with the discussion of our approaches and their limitations (Section 3.6), and conclusions (Section 3.7).

3.1 Overview

The core objectives of this chapter are to analyze, design, and implement work- and CPU-efficient window aggregation operators for single-core execution. In Section 2.3, we discussed that efficient calculation of overlapping windows (i.e., sliding or multi-query workloads) involves two orthogonal techniques to prevent repeated computations: (i) partial aggregation of non-overlapping window chunks [146, 140, 46, 212]; (ii) incremental execution for sharing intermediate aggregates between overlapping windows [106, 194, 4].

Example: Figure 3.1 presents an example of the two approaches. On the left, the data parallelization opportunities (i.e., SIMD) are presented by partial aggregates p_1-p_4, which are CPU-efficient but work-inefficient. On the right, incremental algorithms offer opportunities for work-efficient execution at the expense of parallelism.

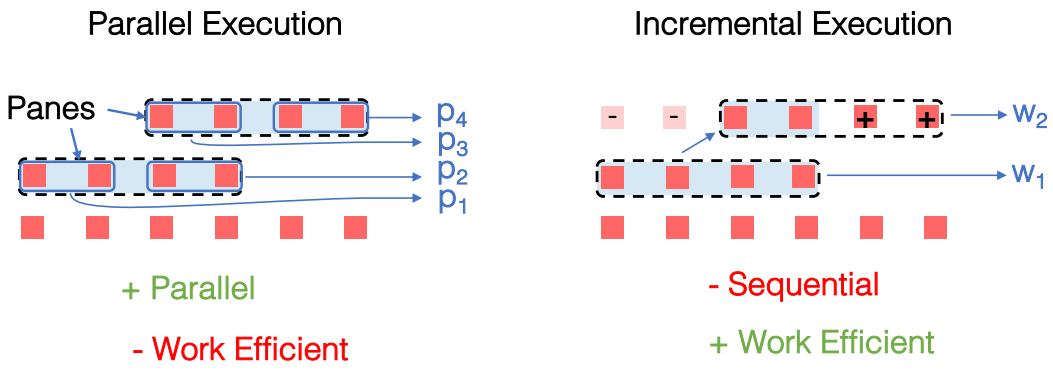


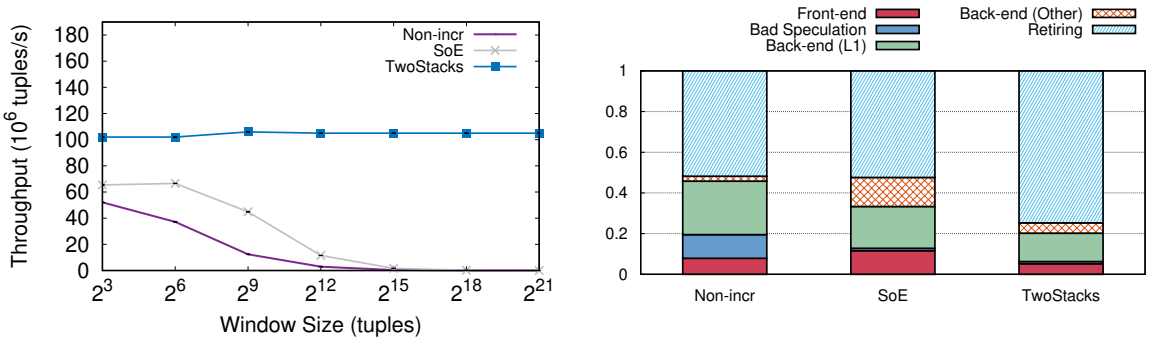
Fig. 3.1 Opportunities for data-parallel and incremental computation for sliding windows

To address the tension between incremental execution and data-parallelism, we provide a family of window aggregation techniques that accelerate partial aggregation while enabling *computation sharing* for overlapping windows. For partial aggregation, we accelerate computation through SIMD-parallelization and introduce a set of novel hardware-conscious optimizations, such as internal data structure decomposition and data minimalism. Our approach is called HammerSlide and resolves requirement **R2** for work- and CPU-efficient execution. SPEs can use HammerSlide to handle both incremental and non-incremental aggregation, resulting in simpler and faster system designs. Regarding the computation sharing opportunities of multi-query workloads (requirement **R3**), we introduce a novel algorithm, called SlideSide, that uses different processing schemes for invertible and non-invertible functions. SlideSide, thus, exploits their algebraic properties to increase work efficiency. Both techniques achieve significantly better performance (requirement **R1**) for window aggregation than existing solutions through careful, hardware-conscious optimizations. In addition, they can be integrated into modern SPEs (e.g., Apache Flink [45] or Apache Spark [231]) as drop-in replacements for any associative aggregation operator. We next outline the key components of our solution.

Performance analysis. We first analyze in Section 3.2 the performance of the three best-performing window-aggregation algorithms for single-query execution from recent literature [106] to identify the characteristics of existing solutions. We show how all approaches – and arguably the entire problem of streaming window aggregation – expose fundamentally different performance patterns than classic relational database problems with various experiments. Window aggregation tends to suffer much less from memory bandwidth starvation and much more from L1-data cache latency and control hazards.

Accelerating partial aggregation. Based on the observations from our analysis, we study the applicability of several optimization techniques for partial aggregation, including the design and implementation of vectorized operators, to address the identified bottlenecks. Furthermore, we demonstrate how these optimizations affect the runtime of window aggregation.

In Section 3.3, we combine the aforementioned optimizations to design HammerSlide, a novel window aggregation technique that extends the best-performing incremental algorithms for sliding window computation [4]. HammerSlide is competitive with a highly optimized non-incremental algorithm, even when processing tumbling windows, and there is no need to share intermediate window results. Besides the potential of simplifying SPEs for both tumbling and sliding windows, HammerSlide yields almost 2× better performance for sliding windows with a slide greater than one tuple (requirement **R1**). Finally, we also conduct experiments inside a state-of-the-art SPE and measure the end-to-end performance of our approach, which yields up to 12× throughput improvement. Since many of the most widely



(a) Throughput with varying window sizes (b) CPU cost breakdown for window size 1024

Fig. 3.2 Evaluating min function with window slide of one

used SPEs are built on a JVM [211, 45, 231], we demonstrate the steps for integrating our C++ implementation efficiently into a Java-based SPE.

Accelerating incremental execution. With respect to result sharing between multiple queries, we use our performance study from Section 2.3.4 to propose a novel approach for incremental processing in multi-queries scenarios called SlideSide (Section 3.4). Our solution extends the logic of TwoStacks [106] using the insight that the algorithm maintains a running *prefix-/suffix-scan* over the input stream (see Section 2.3.3). SlideSide optimizes its performance for associative aggregation functions based on their algebraic properties and can serve as a drop-in replacement for the aggregation operator in an SPE. We demonstrate that SlideSide is competitive with highly optimized single-query algorithms, while it yields up to 2× better throughput and comparable latency in the multi-query scenario (requirement **R1**). After presenting both HammerSlide and SlideSide, we distinguish sections of the window aggregation problem space in which different approaches perform best.

3.2 Revisiting the design of window aggregation

Before designing algorithms that accelerate window aggregation, it is crucial to identify the bottlenecks of existing solutions presented in Section 2.3.4. To simplify the process, we conduct performance analysis for single-query execution using the best-performing algorithms [106] for three different workloads: (i) tumbling windows; (ii) sliding windows with invertible functions; and (iii) sliding windows with non-invertible functions. Next, in Section 3.4, we use our findings as a guideline for accelerating multi-query execution. All algorithms evaluated in this section have $O(1)$ insertion time complexity per tuple and $O(n)$ space complexity, while the window computation time ranges from $O(1)$ to $O(n)$. To make this chapter self-contained, we briefly present these algorithms:

Algorithm 1: TwoStacks algorithm from [106]

```
1  fun query()
2      return Σ_F^⊕ ⊕ Σ_B^⊕
3  fun insert(v)
4      B.push(v, Σ_B^⊕ ⊕ v)
5  fun evict()
6      if F.isEmpty() then                                   ▷ Flip Phase
7          while not B.isEmpty() do
8              F.push(B.top().val, B.top().val ⊕ Σ_F^⊕)
9              B.pop()
10     F.pop()
11 fun Σ_F^⊕
12     if F.isEmpty() then return neutralVal
13     else return F.top().agg
14 fun Σ_B^⊕
15     if B.isEmpty() then return neutralVal
16     else return B.top().agg
```

- The non-incremental algorithm recalculates the aggregate result for each window from scratch. Its time complexity per window slide is $O(n)$, but it does not introduce any overhead for tumbling windows (i.e., disjoint sets).

- Subtract-on-Evict (SoE) [106] is the best-performing approach in the case of invertible functions, e.g., sum. With SoE, the result of the previous window instance is reused to compute the next in constant time by removing the expired tuples and merging the new data. However, SoE cannot efficiently compute non-invertible functions (e.g., min), as the whole window needs to be rescanned in the worst-case scenario with $O(n)$ time complexity (i.e., the previous minimum value is evicted).

- TwoStacks [4, 106] implements a queue using two stacks, a *back* and a *front* stack. As shown in Figure 2.10, each stack element contains a value (white column) and an aggregation of everything below it (blue/green columns). To clarify how TwoStacks works, we provide its pseudocode from previous work [106] in Algorithm 1. The stacks are denoted as `F` and `B` (i.e., *front* and *back*), while `val` and `agg` are the values and aggregates of a stack element. The symbol $\oplus$ denotes a combiner function from Section 2.3.1, and `neutralVal` is its neutral element [202]. TwoStacks relies on three functions for window aggregation: (i) `insert` for adding elements in a window and incrementally computing the aggregate; (ii) `evict` for removing expired elements;

(iii) `query` for computing the current aggregate result. When new elements enter the window, they are pushed on `B` (line 4). Elements that fall off the window are evicted from `F` in line 10. During the eviction, if `F` is empty, the algorithm flips `B` onto `F`, reverting the order of elements. This allows TwoStacks to evict the oldest elements first in constant amortized complexity from the front stack, as the aggregates of the remaining elements are precomputed. To compute the aggregate result of the current window, we must combine the top elements of both stacks in line 2. For a window size of n tuples, TwoStacks requires $2n$ partial aggregates and exhibits $O(1)$ amortized complexity for both invertible and non-invertible functions, as all operations invoke $\oplus$ constant times, on average.

Having introduced the three basic algorithms, we now measure their throughput performance after exploiting possible optimization opportunities, such as storing data in fixed-sized arrays. First, we evaluate them in the most challenging case in terms of repeated computations: a non-invertible min aggregate over a count-based window of slide one tuple. For our experiments, we vary the window size from eight to two million tuples to stress the efficiency of incremental execution. Finally, we generate a stream of random uniformly distributed integers as input (see Section 3.5 for our hardware set-up details).

In Figure 3.2a, we observe that the TwoStacks algorithm is affected negligibly by the window size as expected due to its aggregation time complexity. However, the other two algorithms experience severe performance degradation as the window size increases. For the SoE algorithm, we observe that the cost of computing the new minimum if the current value is evicted increases with the window size and dominates the performance for large windows. The computation is executed by re-scanning the entire window. The decreasing performance of the naïve, non-incremental algorithm is consistent with the explanation from above. Thus, we deduce that for non-invertible functions, the TwoStacks algorithm is the most efficient without applying any hardware-conscious optimizations.

However, the results dramatically change when performing invertible computations, as SoE outperforms the remaining algorithms for small slides, and the non-incremental approach yields the best results for larger slides. In addition, when comparing the throughput results with the per-core bandwidth, as shown in Figure 2.11, we observe a striking gap between memory and processing bandwidth. Therefore, it is unclear how to design an approach that bridges this gap while exhibiting comparable or better performance for different workloads.

To identify which parts of the microarchitecture are the bottleneck, we break down the elapsed cycles by CPU component, as described in Section 2.2.1. For this execution time breakdown, we use as workload the min function over count-based windows with size 1024 tuples and slide one. Figure 3.2b shows that all algorithms spend at least 50%

of their cycles retiring instructions. When calculating a window result, all algorithms perform incremental aggregation,[1] introducing control and data dependencies that affect work efficiency but not CPU efficiency. The execution is back-end bound in most remaining cycles due to L1 cache accesses. This is consistent with our expectations since most data accesses are performed to manage the window state that resides in the L1 cache. Overall, we found that the best performing algorithm, i.e., TwoStacks, spends less than 10% on bad speculation or front-end bound. This indicates two opportunities for performance improvement: (i) improving CPU efficiency with SIMD-intrinsics; and (ii) reducing the number of L1 data accesses. Next, we discuss how we optimize both ends to design HammerSlide.

3.3 Work- and CPU-efficient window aggregation

In this section, we address the bottlenecks identified in Section 3.2 to design a work- and CPU-efficient single-core execution. We propose a family of window aggregation optimizations that focus on data layout, data storage format, data-parallel operations, and compression techniques. For each optimization and design decision introduced, we provide a graph to show its effect on the runtime and evaluate its importance. As a running example for all microbenchmarks throughout this section, we use the computation of the min function for windows of size 1024 and slide 64 tuples while generating a stream of random uniformly distributed integers.

Storing data in circular buffers. Our first optimization is about the data layout of window state, and it applies to both TwoStacks and SoE algorithms. A generic implementation can allow the state of these algorithms to grow arbitrarily, offering the flexibility to process windows that vary in size (e.g., time-based windows). This flexibility, however, comes at the cost of more complex addressing logic and bounds checking (indexed access must dereference two pointers). We avoid these overheads by using circular buffers, allocating sufficient space for count-based windows to fit the entire window. For time-based windows, we allocate space according to generous estimates and resize the buffer during execution if required. The logic of circular buffers allows us to process unbounded data streams by wrapping around to the beginning of the underlying allocated memory and offers a predictable memory access pattern. This data layout is CPU-cache-friendly because tuples can be pre-loaded at a hardware level. Below, we evaluate our structure in comparison to stacks from the C++ Standard Template Library (STL) that use deque as their standard container and require two pointer dereferences.

[1] The non-incremental approach does not perform incremental execution across windows.

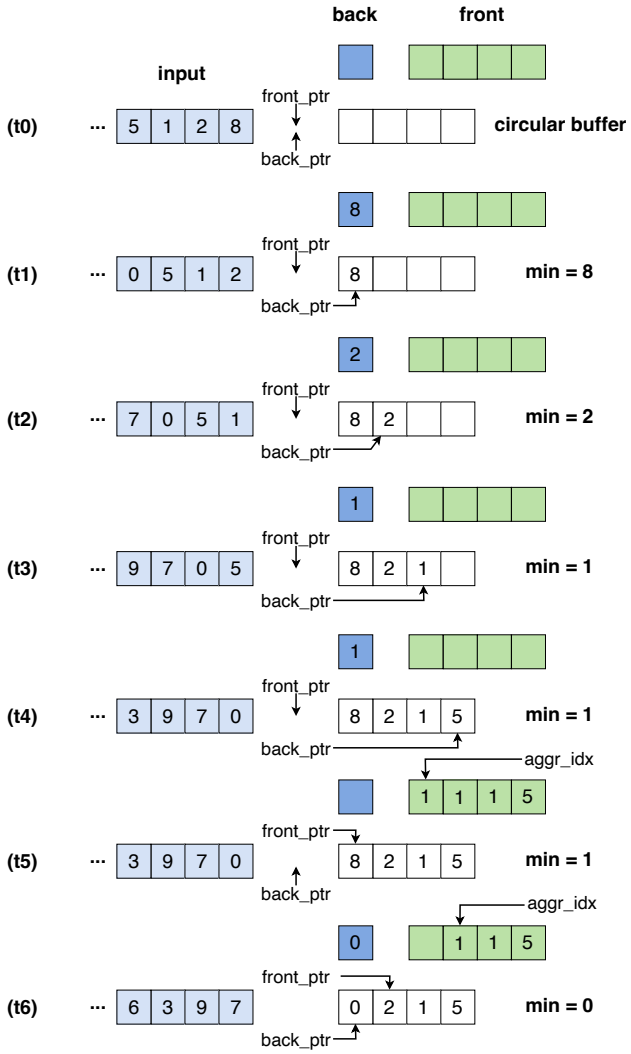


Fig. 3.3 Example of HammerSlide with TwoStacks

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

In Figure 3.3, we present the implementation of the TwoStacks algorithm with a flat circular buffer in the case of min aggregate, using a window of size four and slide one for simplicity. The circular buffer (i.e., white boxes) stores the actual stream values, while the boxes above store the aggregates of the back (blue) and front stack (green). The two stacks are overlayed atop the circular buffer using the respective pointers (i.e., `back_ptr` and `front_ptr`). We move these pointers according to the window semantics to perform the

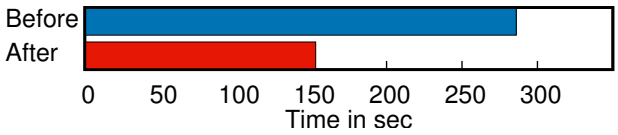


Fig. 3.4 Performance improvement with circular buffers

operations discussed in Section 3.2: insertions to the back stack, evictions from the front stack, or flipping the back stack onto the front. Assuming tuples with an integer value as a payload for simplicity, they enter the system in the order: 8, 2, 1, 5, 0, 7, 9, 3, 6.

In phase (*t1*), 8 is inserted in the back stack, which is reflected by moving the `back_ptr` and writing the value in both the circular buffer and the back stack's aggregate, while the front stack is empty. Next, in phases (*t2*), (*t3*), and (*t4*), we insert new values by moving the `back_ptr` and changing the back stack's aggregate, if necessary, by applying the min function to the previous aggregate and the new value. At this point, the window has four elements, and we are ready to emit the result, but the front stack is still empty. Thus, we have to perform the flip phase (*t5*) of the algorithm shown in Algorithm 1.

For the flip phase, we need to copy the elements from the back stack to the front. As the back stack elements are removed in a LIFO order, their values and aggregates are computed and pushed in the front stack in reverse (i.e., from right to left). However, storing the actual stream values in a circular buffer and using the stack pointers means we do not have to copy data between the stacks. Instead, we traverse the buffer backwards to calculate the front stack aggregates. More specifically, we fill the green boxes from 5 to 8: 5 is the first min as there is no other value yet, and then 1 becomes the min of the remaining aggregates. The `back_ptr` is set to null, as the back stack is empty, the `front_ptr` points at the beginning of the circular buffer, and the `aggr_idx` pointer denotes the aggregate of the front stack's top element.

Finally, in phase (*t6*), we have to evict a tuple in order to insert a 0 into the back stack. Evictions are performed on the front stack by moving the `front_ptr` to the next element (value 2) and the `aggr_idx` to the next aggregate (value 1). Removals do not require additional writes, as both the circular buffer and the front stack aggregates are eventually overwritten by new values without additional overhead.

Avoiding data copies removes the unnecessary writes in the flip phase. Moreover, combined with the contiguous memory's data locality, this optimization significantly improves the performance, reducing the runtime from 286 s to 153.5 s (80%) in Figure 3.4.

Decomposed intermediates & data minimalism. We now examine the data storage format of the window state. As SoE maintains a single accumulator value for each aggregation, we focus on the TwoStacks algorithm that maintains two stack data structures. In particular, each element stored in the stacks has two attributes: the inserted value and the aggregate of

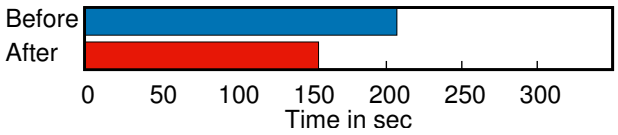


Fig. 3.5 Decomposing intermediates and reducing data

all the values beneath it. It is a fact that these can be stored in n-ary form (also known as struct-of-arrays or, more colloquially, columnar format). While decomposition does not yield an immediate performance benefit (as the application is not memory-bandwidth bound), it enables several subsequent optimizations, which we discuss in the following.

First, we observe that the back and the front stack are used differently (see Figure 2.10). From the back stack, only the value column and the top element of the aggregate column are ever read. The value column is only needed to perform the flip. From the front stack, only the aggregates are read by the **pop** operation. By exploiting the decomposed format of the stacks, we can thus elide the allocation and maintenance of the respective data buffers. Instead, we only store the value column and the top element of the aggregate stack from the back stack and only the aggregates from the front stack. This reduces the number of L1 cache misses since fewer attributes have to be stored in the cache.

Based on these observations, we implement the TwoStacks algorithm as shown in Figure 3.3 and maintain the least possible values that allow us to compute the aggregation result. For window slides greater than one, we realize that it's sufficient to maintain one single value per window slide (also equivalent to a partial aggregate in this case) for the front stack, as all the elements of a specific slide will be evicted simultaneously. These optimizations yield an effective reduction of the runtime by 33% in Figure 3.5.

Bulk insertion & SIMD scanning. To increase CPU efficiency, it is natural to use vector instructions to insert new values into the respective data structures in bulk. We also pre-aggregate the values upon insertion for slides greater than one (i.e., partial aggregates), implying reduced output granularity. Since we have replaced the stacks with circular buffers in the TwoStacks approach, this optimization naturally applies to SoE too. For example, in the insertion phases like (*t2*), (*t3*), (*t4*), and (*t6*) of Figure 3.3, if the slide is large enough, we can apply our aggregation functions with SIMD intrinsics on each slide to compute the running partial aggregate inserted in the back stack.

The main benefit of decomposed storage introduced before is that it enables SIMD instructions for scanning the buffers when calculating aggregates. We implement a SIMD-parallel version for the standard SQL aggregation functions (min, max, sum, count, avg) and many statistical properties (e.g., standard deviation) using AVX-256 bit compiler intrinsics. More precisely, in the case of the SoE algorithm, we apply these parallel versions during

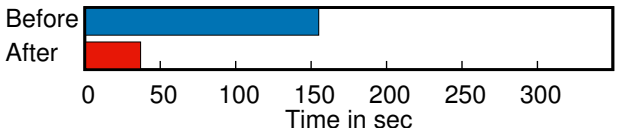


Fig. 3.6 Performance with bulk insertion and SIMD scanning

the eviction phase when we want to recompute the running value of the aggregation. For the TwoStacks approach, we utilize this optimization within the flip phase, as we show in (*t5*) of Figure 3.3, in which we compute a single aggregate value per slide. These optimized functions for scanning the buffers reduce the runtime from 155.6s to 37s (i.e., 4.2× speedup), as shown in Figure 3.6.

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

Both SIMD scanning and bulk insertion apply to aggregation functions other than the ones we implement in this work. Their combination introduces parallel processing within a partial aggregate (i.e., pane) for single-core execution and is our optimizations' most important improvement factor.

Stack RLE compression. In Figure 2.10b, we notice that the aggregate values on the front stack[2] expose an exploitable pattern: since each slot contains the minimum of the slot beneath it and its aligned value, the probability is high that the value in a slot is equal or smaller to the value in the slot beneath it. This data pattern is naturally amenable to run-length encoding (RLE) [189]. Instead of inserting a new value and increasing the top pointer, only the top count must be increased. This reduces the size of the stack and allows us to make fast comparisons and scan the front-stack using SIMD instructions.

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

Discussion. As presented in the performance analysis above (Section 3.2), for designing hardware-efficient aggregation operators, we must improve the CPU efficiency and reduce the L1 accesses of existing solutions [4]. However, while incremental algorithms are work-efficient (see the retiring instruction cycles in Figure 3.2b), it is challenging to increase their CPU efficiency due to their inherent data and control dependencies. For that reason, we contribute a set of novel optimizations that can transform existing work-efficient incremental

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

algorithms, such as TwoStacks or SoE, to both work- and CPU-efficient approaches. The following optimizations lead to the design of hardware-conscious operators: (i) cache-friendly storage data layout; (ii) removal of redundant storage requirements that lower the L1 cache misses; and (iii) CPU-efficient execution with data-level parallelism (SIMD instructions). As future work, we want to investigate how to apply these optimizations to other window operators (e.g., joins [126]).

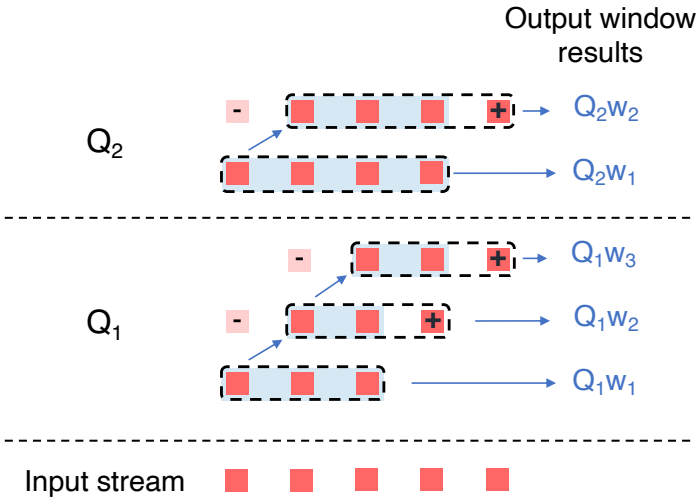


Fig. 3.7 Answering two concurrent queries over the same data stream

3.4 Efficient incremental aggregation for multiple queries

Having addressed the challenge of work- and CPU-efficient computation for single-query execution, we now focus on accelerating incremental aggregation in a multi-query environment. For such workloads, it is important to identify result-sharing opportunities to avoid redundant work and reduce end-to-end latency.

Example: As shown in Figure 3.7, when answering two queries Q_1 and Q_2 with windows of size three and four and slide one tuple, there is an overlap of intermediate results due to their window specifications. This overlap results in performing redundant operations that can be avoided upon updating the window contents (the minus and plus symbols denote removing and adding a tuple in a window). For example, in this scenario, if we compute the first window result of Q_1 (i.e., Q_1w_1), we can then compute the first window of Q_2 (i.e., Q_2w_1) by simply adding the fourth tuple from the input stream. We can, therefore, reduce the number of aggregations by two, as we do not need to compute Q_2w_1 from scratch.

However, with the increasing number of concurrent queries, sharing efficiently intermediate results between different windows becomes more challenging. To address this challenge, we contribute SlideSide, a novel incremental algorithm that aggressively reuses intermediate results among concurrent queries. Instead of processing invertible and non-invertible functions with a uniform approach similar to HammerSlide though, we decide to distinguish SlideSide's execution strategies. SlideSide uses different processing schemes based on the invertibility property to eliminate redundant operations in multi-query workloads. Regarding the algebraic properties of the aggregate functions, SlideSide has the same requirements as the state-of-the-art algorithms described in Section 2.3.4 (associative aggregate functions) and can be applied to FIFO windows (i.e., in-order data). Fundamentally, SlideSide is an extension of the TwoStacks algorithm that exploits the design principles introduced in Section 3.3 (e.g., data structures with sequential memory layout).

In the remainder of this section, we discuss how SlideSide employs different processing strategies for invertible (Section 3.4.1) and non-invertible (Section 3.4.2) functions to speed up incremental execution.

3.4.1 Result sharing for multiple invertible aggregates

The most straightforward case is applying the same invertible aggregation functions, such as sum, over a data stream. The natural approach of evaluating simultaneous windows would be to run multiple loop-fused instances of SoE, the best performing algorithm for invertible functions. However, we found that we can extend the TwoStacks algorithm to support the workload of concurrent queries as well, yielding a more cache-efficient approach similar to what we discussed in Section 3.3. Somewhat surprisingly, we implement this using only two stacks (illustrated in Figure 3.8). Like the single query case, the elements of the back and front stacks share the same memory space, while their aggregates are kept separately. Next, we will explain the algorithm, and then we will present an example execution using the two queries shown in Figure 3.7.

During the initialization phase of Algorithm 2, the *back* stack, the *front* stack and a circular buffer of *elements* are allocated with size equal to the largest window from a given set of queries (Q), and initialized with the neutral element [202] of the aggregate function (lines 1-4). For every input value *val* from the stream, we call the *insert* function and compute the results for every query in Q with *emitResults* in lines 6-8.

Upon the arrival of a new element, using the *insert* function (Algorithm 3), its value is stored in the next available slot of the circular buffer, defined by the *curPos* variable in line 4. The back stack is used for maintaining the *prefix-scan* of the input with every insertion (line 5), similar to Section 3.3. Suppose we reach the end of the circular buffer. In that case,

Algorithm 2: SLIDESIDE (INV) PSEUDOCODE

Input: A set of aggregate queries Q, a combine operation $\oplus$, an inverse operation $\ominus$
Output: The results of the window queries in Q

```
1 windowSize ← Q.getMaxWindowSize()
2 backStack[windowSize + 1] ← {neutralVal}        ▷ used for prefix-scan
3 frontStack[windowSize + 1] ← {neutralVal}       ▷ used for suffix-scan
4 elements[windowSize] ← {neutralVal}             ▷ used for input stream
5 curPos ← 0
6 for val ∈ stream do
7     insert(backStack, frontStack, elements, curPos, windowSize, val)
8     emitResults(backStack, frontStack, curPos, windowSize, Q)
```

Algorithm 3: Algorithm for insert(...) – SLIDESIDE (INV)

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

we wrap around to the beginning and compute a *suffix-scan* over the input (lines 2-3) before applying the new insertion, which guarantees correct results after evictions. The evictions are performed by overwriting obsolete data as in Section 3.3. Finally, note that, as in TwoStacks, the computation of the *suffix-scan* occurs infrequently, and the algorithm, thus, yields $O(1)$ amortized complexity.

After the insertion, the *emitResults* function (Algorithm 4) is called for computing the results for each query with the set Q. Based on the values of *curPos* and the window size of each query, this algorithm first computes the `startPtr` and `endPtr` pointers (lines 2-10) that are used for "bookkeeping". Next, we distinguish three different cases based on the value of these pointers: **(a)** if the `startPtr` is 0, the result is already computed by the *prefix-scan* and returned in line 12; **(b)** if the `startPtr` is greater than 0 and the `endPtr` does not wrap around the beginning of the circular buffer, the result value is computed by applying the *inverse* function ($\ominus$) on the values from the back stack in the positions of `startPtr` and `endPtr` (line 16); **(c)** if the `endPtr` has wrapped, the result is computed by *combin*ing ($\oplus$) `backStack[endPtr]` and `frontStack[windowSize-startPtr]` in line 14.

In Figure 3.8, we present an example of SlideSide for the windows from Figure 3.7: Q_1 and Q_2 with size three and four respectively, and slide one. The red boxes represent the result of the queries (i.e., sum function) on each phase. The white boxes hold the values of the start

Algorithm 4: Algorithm for emitResults(...) – SLIDESIDE (INV)

```
1  for query q : Q do
2      curWindowSize ← q.getSize()
3      hasWrapped ← false
4      endPtr ← curPos
5      if endPtr = 0 then
6          endPtr ← windowSize
7      startPtr ← endPtr − curWindowSize
8      if startPtr < 0 then
9          hasWrapped ← true                ▷ window wraps around the circular buffer
10         startPtr ← startPtr + windowSize
11     if !hasWrapped&&startPtr = 0 then
12         res ← backStack[endPtr]                ▷ use the result from prefix-scan
13     else if hasWrapped then
14         res ← backStack[endPtr] ⊕ frontStack[windowSize − startPtr]
15     else
16         res ← backStack[endPtr] ⊖ backStack[startPtr]
17     forward answer res to query q
```

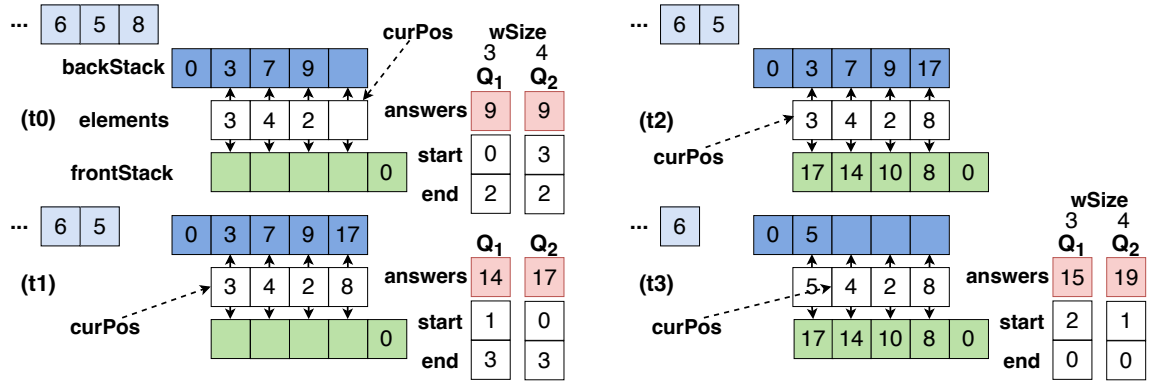


Fig. 3.8 Example of the SlideSide (Inv) algorithm

and end pointers we discussed above. At our initial phase *t0*, the values 3, 4 and 2 have been already inserted in the *elements* buffer (from left to right) and their *prefix-scan* is computed in the back stack above (3, 7, 9). In the next phase *t1*, we have an insertion in the last slot of the *elements* buffer, which triggers the computation of the *prefix-scan* for `backStack[4]` by combining 9 (previous result) and 8.

After the insertion, the algorithm emits results for both queries. For Q_2, the result contains all the elements and the window starts from position 0 (case **(a)** from above). Thus the result is already computed by the *prefix-scan* and can be obtained by accessing the value

of `backStack[3]` which is 17. For Q_1, the result contains only the latest three elements and the window points at positions 1 and 3 (case **(b)**). Thus, the result is computed by applying the *inverse* function on the elements from the back stack placed at that positions.

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

Algorithm 5: SLIDESIDE (NON-INV) PSEUDOCODE

Input: A set of aggregate queries Q, a combine operation $\oplus$
Output: The results of the window queries in Q

```
1  windowSize ← Q.getMaxWindowSize()
2  flipRange ← Q.getMinWindowSize()
3  backStackVal ← neutralVal                          ▷ used for prefix-scan
4  frontStack[windowSize] ← {neutralVal}              ▷ used for suffix-scan
5  elements[windowSize] ← {neutralVal}                ▷ used for input stream
6  curPos ← 0 for val ∈ stream do
7      insert(backStackVal, frontStack, elements, curPos, windowSize, val, flipRange)
8      emitResults(backStackVal, frontStack, curPos, windowSize, Q)
```

Algorithm 6: Algorithm for insert(...) – SLIDESIDE (NON-INV)

```
1  if curPos%flipRange = 0 then                       ▷ compute suffix-scan
2      tmp ← neutralVal
3      inIdx ← curPos
4      for outIdx ← (windowSize − curPos − 1), ..., windowSize do
5          tmp ← neutralVal
6          tmp ← elements[inIdx] ⊕ tmp
7          frontStack[outIdx] ← temp
8          inIdx ← inIdx − 1   ▷ we can stop when encountering the again same value
9      backStackVal ← neutralVal
10 elements[curPos] ← val
11 backStackVal ← elements[curPos] ⊕ backStackVal
12 curPos ← (curPos + 1)%windowSize                   ▷ wrap around the circular buffer
```

Algorithm 7: Algorithm for emitResults(...) – SLIDESIDE (NON-INV)

```
1 for query q : Q do
2     curWindowSize ← q.getSize()
3     pos ← curPos − curWindowSize + 1
4     if pos < 0 then
5         pos ← pos + windowSize
6     res ← backStackVal ⊕ frontStack[windowSize − pos − 1]
7     forward answer res to query q
```

3.4.2 Result sharing for multiple non-invertible aggregates

Let us now discuss the logic behind processing multiple non-invertible functions, illustrated in Algorithm 5, which is similar to Algorithm 2. Yet, the *suffix-scans* have to be triggered more frequently, and we have to keep track of the smallest size from queries Q. Next, the insertion algorithm for non-invertible functions is presented in Algorithm 6. The intuition behind insertion is that, while each of the queries maintains and operates on its own pair of stacks, these can be overlayed and start at the same memory address. In effect, the smallest stack is stored in the same memory region as the bottom part of the next larger, and so forth. To preserve correctness among the query results, the computation of the *suffix-scan* is triggered every time the query with the smallest window size starts to evict (denoted with the `flipRange` variable).

The non-invertible execution can be further optimized, similar to the single-query evaluation in Section 3.3 by maintaining only the top value of the back stack. Regarding performance challenge of frequently triggered *suffix-scans* based on the smallest window size in Algorithm 6, we observe that we can reduce the number of updates required. During the *suffix-scan* computation, if the algorithm finds the same partial aggregate twice, it can stop propagating the changes from the current position until the end of the front stack, as the remaining partials hold already the correct value. This dramatically reduces the overhead of multiple flip phases and results in constant amortized complexity, as shown in Table 2.1.

Finally, the *emitResults* function (Algorithm 7) illustrates how SlideSide answers multiple non-invertible queries. The process is simpler than invertible functions: for each query, SlideSide always uses the back stack with the appropriate value from the front stack.

Discussion. To simplify the description of SlideSide, in Algorithms 2 and 5, we assumed that all window semantics have a slide of one. For slides greater than one, HammerSlide can be used to accelerate partial aggregation, while SlideSide performs incremental execution over the partial aggregates for work efficiency. In addition, both Algorithms 2 and 5 must change to keep track of when the *emitResults* is called similar to previous work [212, 194]. Finally,

when answering queries with invertible and non-invertible functions, SlideSide overlaps the stacks of both types and updates partial results accordingly.

3.5 Evaluation

To evaluate the benefits of our solutions,[3] we conduct experiments against the state-of-the-art incremental algorithms using synthetic and real-world datasets described in detail along with our evaluation set-up in Section 3.5.1. First, we explore the robustness of different single-query algorithms based on the window slide, number of recomputations due to data evictions, and window size and explore the efficiency of HammerSlide for time-based windows (Section 3.5.2). Then, we integrate HammerSlide with an SPE (Section 3.5.3) to perform an end-to-end performance evaluation. Finally, we evaluate SlideSide for both multi-query and single-query execution (Section 3.5.4) and advise which approach to use based on the workload characteristics (Section 3.5.5).

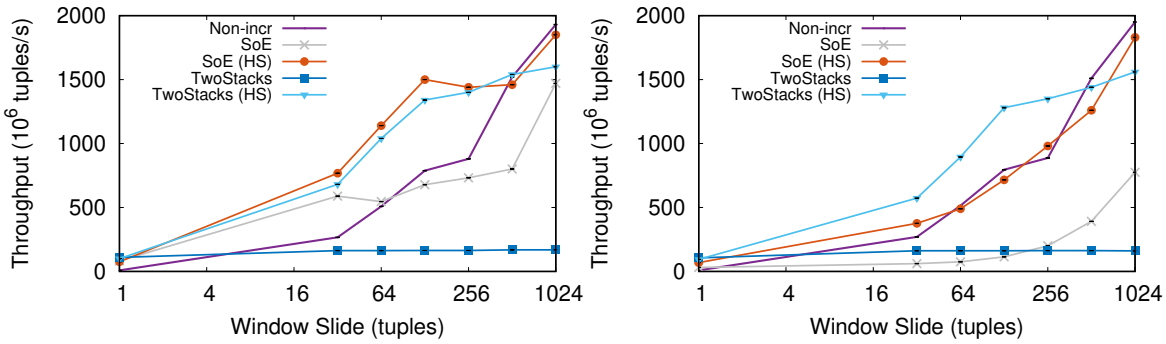
3.5.1 Experimental set-up and workloads

Hardware. 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 evaluate all approaches with single-threaded execution. To achieve the minimum NUMA interference, we bind our experiments to processor 0 (on NUMA node 0). The same applies to all our memory allocations, which were bound on that NUMA node.

Workloads. For our microbenchmark evaluation, we generate data streams of 32-bit integer values drawn from a uniform distribution. On this input dataset, we evaluate min and avg, which are representative of the two classes of functions we study. We identify two separate experiments: (i) we keep the slide constant at one while changing the size; and (ii) we keep the size constant at 1024 tuples and alter the slide from one until the window becomes tumbling.

For the macro-evaluation, we study two workloads: (i) one that emulates a cluster management scenario; and (ii) one that emulates an anomaly detection scenario. The first dataset represents a trace of time-stamped measurements taken from an 11,000-machine Google's cluster [220]. Each tuple contains information about metrics related to tasks executed on the cluster, such as CPU utilization or task priority. On that dataset, we evaluate

[3]The source code is available at https://github.com/grtheod/Hammerslide (2K lines of C++14).



(a) Throughput for invertible functions (avg) (b) Throughput for non-invertible functions (min)

Fig. 3.9 Count-based windows with window size of 1024 tuples and variable slide

a query that expresses a common cluster monitoring task [129] and reports the global average requested CPU utilization of submitted tasks (i.e., without a `GROUP-BY` clause). The second dataset uses the energy consumption trace from a smart electricity grid containing smart meter data from households' electrical devices in households [123]. We use two queries to perform analysis over the stream and detect outliers: $\mathsf{SG}_{\mathsf{sum}}$ and $\mathsf{SG}_{\mathsf{min}}$ that compute a sliding global sum and min respectively over the meter load (see SG_1 from Section A.1 for reference).

To avoid any possible network bottlenecks, we generate ingress streams in memory by pre-populating buffers and replaying tuples continuously.

Metrics. The main performance metrics considered in the following benchmarks are throughput and end-to-end latency. We define throughput as the average number of tuples processed within a time unit (e.g., one second). The end-to-end processing latency [217] is defined as the difference between the time when a tuple enters the system and when a window result is produced. Candlesticks in plots show the 5^{th}, 25^{th}, 50^{th}, 75^{th} and 95^{th} percentiles.

3.5.2 Studying workload characteristics for single-query execution

We measure the impact of different workload characteristics, such as window semantics and aggregation types, on the performance of HammerSlide for single-query execution. We compare HammerSlide applied on TwoStacks and SoE with their unoptimized versions and the non-incremental approach that performs best for tumbling windows.

The impact of window slide. To study the effect of the slide on the throughput of the aforementioned approaches, we use avg and min in Figures 3.9a and 3.9b respectively. For both functions, we observe that the plain TwoStacks algorithm performs robustly but almost across the board worst of all our studied approaches. For avg, the plain SoE is quite

competitive and outperforms the non-incremental algorithm for slides less or equal to 64. However, we find that both algorithms with HammerSlide optimizations perform significantly better than their unoptimized counterparts: up to 11× better in the case of the TwoStacks algorithm. For min, the SIMD-enabled TwoStacks algorithm outperforms all others (by almost 80%) for slides less than half of the window size and is never more than 10% worse than the best algorithm. Given the intricate nature of the implementation and the fact that there is no potential for results reuse to exploit, one might expect worse behaviour due to the complex CPU instruction stream. However, it turns out that our careful optimization yielded an implementation with very little overhead.

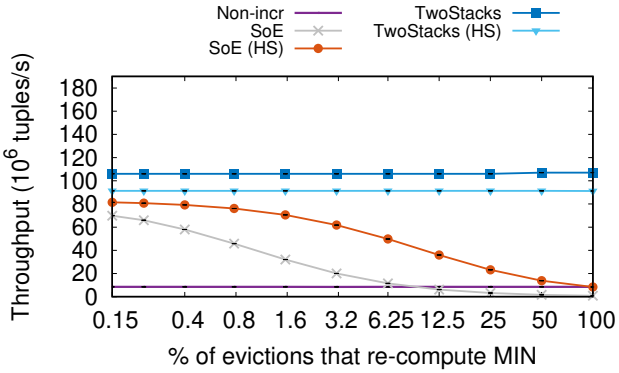
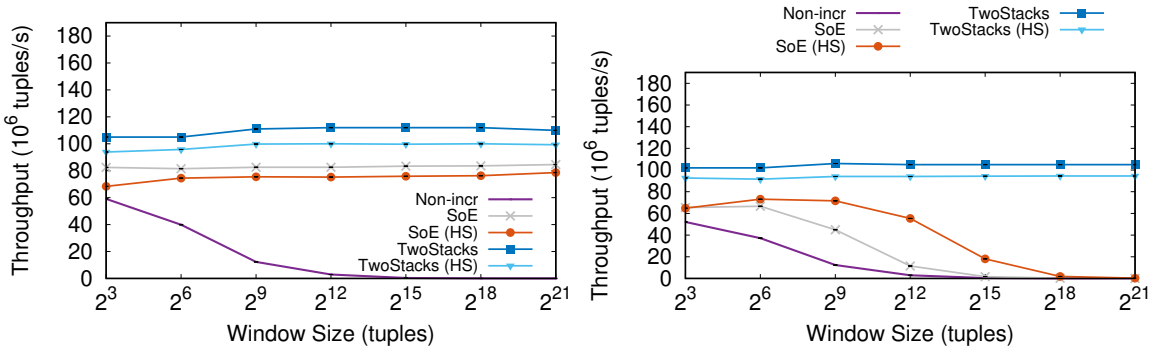


Fig. 3.10 How evictions influence throughput (min with window size 1024 and slide 1)

The impact of window re-computations. When comparing the SoE to the TwoStacks approach for min, the advantage of TwoStacks lies in the robustness against adversarial input distributions: in the worst case, every eviction causes a re-computation of the window for SoE. To quantify the problem, we study the performance degradation of the different algorithms with the rescan rate (i.e., the percentage of evictions that cause a re-compute) by setting the window size to 1024 and the slide to one tuple. Further, we restrict the number of unique values in our input distribution to force more re-computations and present the results in Figure 3.10. We observe that both SoE implementations experience severe performance degradation when the re-compute rate increases. Figure 3.10 shows that the TwoStacks implementations are robust against that input distribution.

The impact of window size. Next, we vary the size of a count-based window with a constant slide of one and consider again two aggregates, avg (Figure 3.11a) and min (Figure 3.11b). In both cases, for all window sizes, the TwoStacks algorithm has the highest throughput at approximately 110 million tuples/s. Since the slide is 1, the optimized version of TwoStacks is slightly slower because of the overhead of checking for opportunities to vectorize code, but in this case, there are none.



(a) Throughput for invertible functions (avg) (b) Throughput for non-invertible functions (min)

Fig. 3.11 Count-based windows with variable window size and slide one tuple

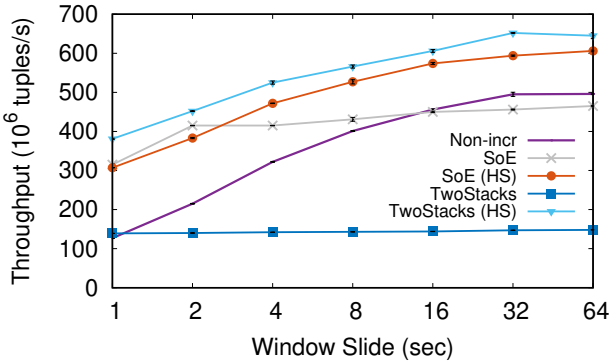


Fig. 3.12 Time-based 64-seconds windows over the Google cluster stream (avg)

For min (Figure 3.11b), the throughput of TwoStacks is not affected by the frequent evictions of the minimum value from the current window, as it recomputes the result in constant time. Note that the percentage of inserts that evict the current minimum is 6.25%. All other algorithms are affected by the growth of the window size: the overhead of recomputing min from scratch over the entire window dominates and causes a significant throughput drop, up to 100×. Only the optimized SoE algorithm can cope with that overhead, achieving only a 2× slow-down compared to TwoStacks when the window size is less than 4096 tuples because it benefits from vectorized instructions to re-scan the window.

For avg (Figure 3.11a), the superior performance of TwoStacks over SoE is less pronounced: it is only 1.5× faster. The non-incremental algorithm experiences a similar fall in throughput as in the case of min. The non-incremental algorithm can be up to 300× slower than TwoStacks when the window size is 32K tuples.

The impact of time-varying window semantics. Next, let us examine how HammerSlide optimizations apply to time-based window queries using the Google's cluster trace. This use case differs from the previous workloads, as both window size and slide are now defined by

the time dimension instead of the number of tuples. Noteworthy is that the load exposed by the trace is very spiky: the event rate varies from 0 to 100K events per second. Thus, this benchmark significantly stresses all implementations' ability to cope with variable window semantics and changes in the workload during runtime. On this dataset, we evaluate avg with a time-based window of 64 seconds and a varying window slide from 1 to 64 seconds.

Figure 3.12 shows that the plain TwoStacks algorithm does not perform well on time-based windows: its throughput, steady at roughly 150 million tuples/s for all slides, is hindered by frequent, unoptimized flipping the front stack to the back. The non-incremental algorithm, on the other hand, outperforms TwoStacks when the slide is greater than 1 second and matches the throughput of SoE when the slide is 16 seconds or higher. This can be explained by the fact that slides from 16 onwards contain a big percentage of the window data because of the skewed input data distribution; thus, recomputing a window result from scratch imposes no extra computation overhead—in fact, it is even cheaper than maintaining state for SoE. Both versions of TwoStacks and SoE with HammerSlide optimizations perform best, achieving a speed-up between $1.2-4.3\times$. These benefits are due to vectorized instructions and the re-design of the flip operation in TwoStacks. Similarly, the SoE algorithm benefited from avoiding redundant memory copies.

With the time-based windows, though, we observe that the non-incremental approach is not the best performing for tumbling windows, as it was before. This occurs because the compiler can not optimize the code as it did in the previous micro-benchmarks, in which we provided the fixed window size and slide in advance. Thus, HammerSlide presents robust performance and can deal with variations in the workload compared to the other approaches.

Latency results. Both TwoStacks implementations exhibit median latency lower than 15 nanoseconds. They maintain such a low median latency but exhibit periodic latency spikes caused by the flip phase, as we expected. SoE has comparable latency in the case of invertible functions, but without the latency spikes. For non-invertible functions, the latency is affected negatively by the number of evictions (see the impact of evictions above).

Discussion. To sum up, let us briefly discuss a final performance analysis while isolating the effects from an SPE: the by-CPU-component breakdown of all presented approaches for a tumbling window. This breakdown allows us to assess if there is still untapped performance potential in the implementations. By analyzing the cost components of tumbling windows with a size of 1024 tuples, we want to understand the difference in performance between all the algorithms. In Figure 3.13, we witness that the best performing approaches (non-incremental AND HammerSlide) become nearly 30% DRAM bounded, which indicates that they approach the main memory bandwidth limits. This is consistent with our observations

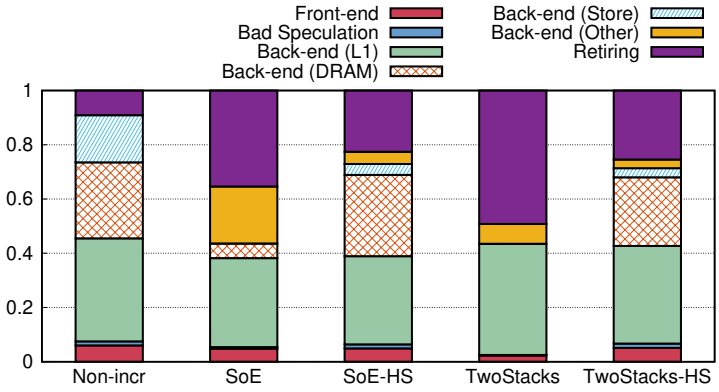
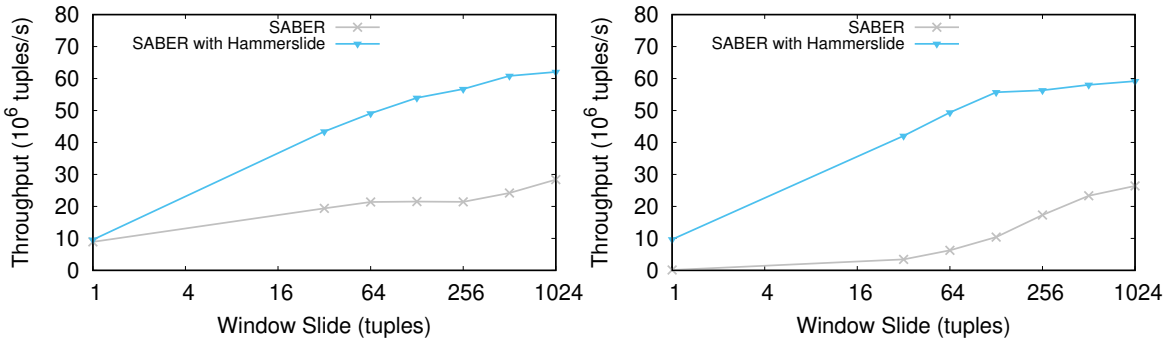


Fig. 3.13 Cost breakdown by CPU components for a tumbling window of size 1024



(a) Throughput for invertible functions (avg) (b) Throughput for non-invertible functions (min)

Fig. 3.14 Integrating HammerSlide with SABER (window size 1024 and variable slide)

in Figure 3.9b in which the tumbling window throughput reached almost 2 billion tuples (or 8 GB) per second.

Overall, our conclusion from Figure 3.13 is that HammerSlide has transformed the performance profile of the plain SoE and TwoStacks algorithms to be closer to that of the non-incremental algorithm, which is the best performing for fixed window sizes and slides: strongly dominated by memory-bandwidth but only little by computational resources (retiring instructions). Thus, our optimized approaches bridge the gap between sliding and tumbling windows and present a significant improvement in throughput compared to the rest, with more than 1.6× greater throughput. In particular, the TwoStacks algorithm with HammerSlide optimizations exhibits comparable or better performance across all examined workloads.

3.5.3 Integrating HammerSlide with a Java-based SPE

For our end-to-end evaluation, we choose to use SABER [137], an SPE that utilizes a hybrid processing model on heterogeneous processors within a single node. In SABER, data is represented in a row-oriented format and stored in a circular buffer, which does not allow us to directly integrate our operators' logic. We extended SABER to support a columnar input format by decomposing streams into multiple circular buffers based on their attributes.

During the dispatch stage of SABER, the system splits the input stream into fixed-sized tasks that may include fragments from multiple windows. Whenever a task is created, we keep track of the respective buffers' offsets according to the data type and dispatch data to the workers. Each worker computes the window boundaries before processing the data during a task's execution. We leverage this approach to compute the partial aggregates of open, closed, or pending windows (see [137]) incrementally within a task. Finally, we utilize the Java Native Interface (JNI) calls and the Java NIO Direct Buffers to avoid unnecessary copies from the Java heap memory when accessing data with our C++ implementation.

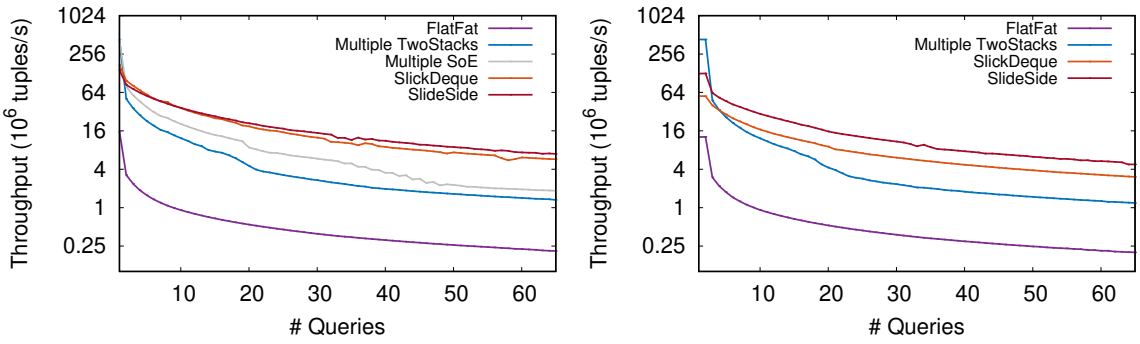
In Figures 3.14a and 3.14b, we evaluate the performance of HammerSlide-optimized TwoStacks algorithm for both invertible and non-invertible functions. SABER uses unoptimized SoE for invertible functions and does not support incremental computation for the non-invertible ones. Both figures illustrate how the bottlenecks change in this end-to-end evaluation, causing lower performance results than the micro-benchmarks.

Figure 3.14a shows that for avg, when the slide is greater than one, the vectorized TwoStacks approach outperforms the baseline up to 2.7×. For min (Figure 3.14b), we observe more than 78× performance benefit for slide one; this is the worst case for SABER as it has to recompute every partial aggregate from scratch. When the slide is greater than one, the SIMD-enabled TwoStacks demonstrates up to 12.2× speed-up for slide equal to 32. When we increase the window slide, the speed-up drops to 2.2× for tumbling windows.

3.5.4 Evaluating multi-query incremental algorithms

In this section, we measure the performance efficiency of SlideSide for multi-query execution and its overhead compared to single-query algorithms. We compare SlideSide against TwoStacks, SoE, FlatFAT, and SlickDeque (see Section 2.3.4) for windows with slide one, for a fair comparison without the HammerSlide optimizations.

In the multi-query experiments, we generate queries of uniformly random window sizes (within the range [1, 32K] of tuples) while maintaining a constant window slide of 1 tuple for all. In this setup, we created workloads that contain from 1 up to 65 concurrent queries. TwoStacks and SoE can not evaluate multiple queries, so we replicate their data structures



(a) Invertible functions (SG_{sum}) (b) Non-invertible functions (SG_{min})

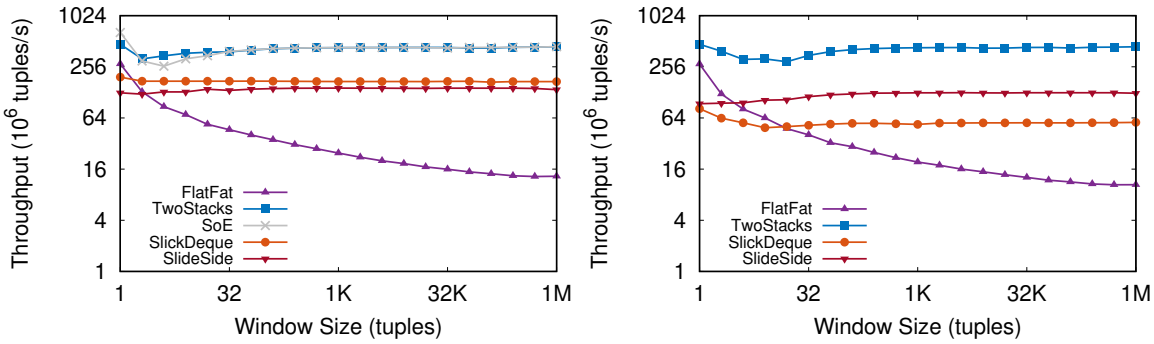
Fig. 3.15 Evaluating SlideSide for multi-query throughput

for every single window definition, as illustrated in Table 2.1. For the performance overhead of SlideSide in single-query workloads, we use queries SG_{sum} and SG_{min} to measure the throughput and latency of different approaches.

Multi-query invertible functions. For invertible functions, we are computing SG_{sum} over different window definitions. Figure 3.15a demonstrates that SoE is the fastest algorithm and outperforms the multi-query solutions by up to 2.5× for a single query. However, as the number of queries increases, the overhead of maintaining multiple data structure replicates becomes noticeable. Thus, we observe that the multi-query algorithms perform nearly 4× better in throughput. Comparing SlideSide with SlickDeque reveals a small performance benefit that reaches up to 40% with the increase of query concurrency. SlideSide's approach allows the compiler to generate more efficient code because of the simpler CPU instruction stream while providing more predictable memory access.

Multi-query non-invertible functions. For the non-invertible functions, we are computing SG_{min} over the generated windows. In Figure 3.15b, we observe that the multiple instances of TwoStacks outperform both SlideSide and SlickDeque for the first two and three workloads respectively. After that point, SlideSide is from 70% up to 2.2× faster compared to SlickDeque and more than 4× compared to the other two techniques in terms of throughput. This illustrates that even though SlideSide requires more memory compared to SlickDeque, its CPU-cache-friendly data layout scales better with the number of queries in comparison to the deque data structure.

Single-query throughput. For this experiment, we use SG_{sum} and SG_{min} over windows with window sizes that vary between one and $1M$ tuples. Figure 3.16a illustrates the throughput penalty introduced by our algorithm for invertible functions in a single query scenario. SlideSide exhibits throughput nearly 3× worse than SoE and TwoStacks. In Figure 3.16b, we observe that TwoStacks is the best-performing non-invertible algorithm for different window



(a) Invertible functions (SG_{sum}) (b) Non-invertible functions (SG_{min})

Fig. 3.16 Evaluating SlideSide for single-query throughput

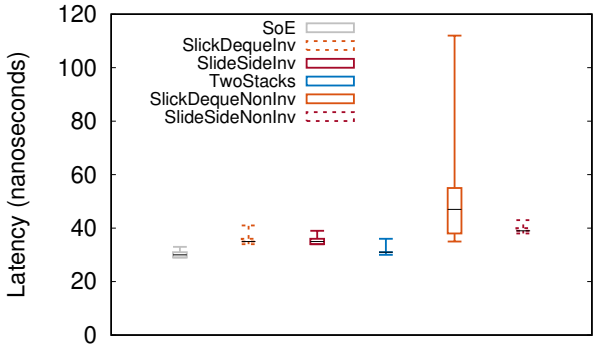


Fig. 3.17 Latency of incremental algorithms in nanoseconds

sizes. In contrary, SlideSide is 3× worse but exhibits better performance than SlickDeque, because of its underlying data structure's sequential memory layout.

Latency comparison for single-query execution. To measure the latency of all the previous approaches, we use a fixed window size of $32K$ and slide of one tuple. In Figure 3.17, we omit the latency of FlatFAT, as it consistently is an order of magnitude higher than the other algorithms. We show that SlideSide exhibits latency that is comparable to the best-performing solutions for both invertible and non-invertible functions (minimal overhead) and better compared to the other multi-query solution, i.e., SlickDeque.

Discussion. Overall, we observe that SlideSide outperforms the remaining approaches up to 2.2× in multi-query workloads and scales better with the number of concurrent workloads. However, for single query evaluation, SlideSide results in nearly 3× worse performance in throughput with comparable latency against the best-performing approaches. This results from the memory pressure of maintaining extra dependencies (not needed by a single query) and a more complex CPU instruction stream that hinders optimizations.

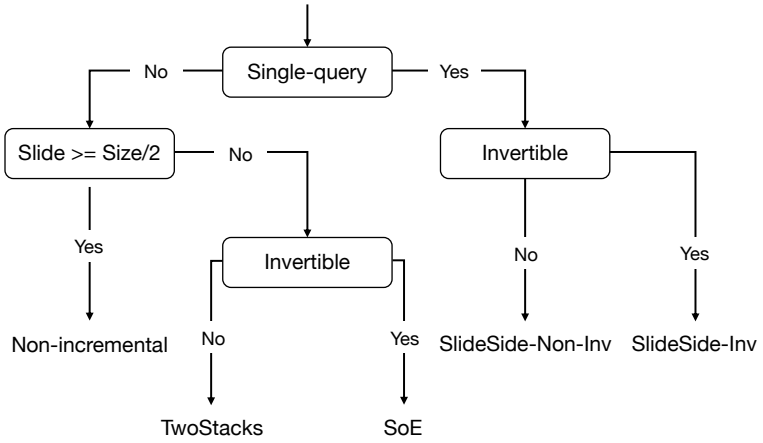


Fig. 3.18 Decision tree for selecting an aggregation algorithm based on different workload characteristics: all approaches (i.e., from Non-incremental to SlideSide-Inv) use the optimizations discussed in Section 3.3 to accelerate the tuple aggregation to partial aggregates when the slide is greater than one.

3.5.5 Deciding based on the workload characteristics

We summarize our observations using the decision tree of Figure 3.18: we propose a different aggregation algorithm that performs best or is comparable to other approaches based on the workload characteristics. Given that the HammerSlide optimizations are beneficial when the window slide is greater than one (i.e., the common case), we assume that all algorithms shown in Figure 3.18 utilize them for partial aggregation (even the Non-incremental approach).

First, we notice from Figures 3.9a and 3.9b that when the slide is equal or greater to half the window size, it is best to recompute the window results from scratch. For single-query invertible functions, SoE exhibits the best throughput, while for non-invertible ones, TwoStacks is the optimal choice. However, as we observe in all micro-benchmarks, it is also reasonable to use TwoStacks with HammerSlide for both cases without noticing significant performance degradation. In the case of multi-query workloads, SlideSide outperforms the other solutions, except for cases with less than three concurrent queries, when SlickDeque or TwoStacks perform better. To simplify the decision tree, we omit the relationship between the window slide and the size for multi-query workloads. While single-query algorithms (i.e., SoE, TwoStacks, Non-incremental) may yield better performance results depending on the window slide, their implementation requires data replication and does not scale with many concurrent queries as shown in Section 3.5.4. It is, therefore, better to use SlideSide regardless of the window slide.

Figure 3.18 shows that an SPE will perform poorly for different workloads or would have to maintain all these algorithms using a cost model. However, this cost model can become

more complicated than our decision tree when more workload variables affect performance. We address this challenge in Chapter 4 by generalizing the aforementioned algorithms with a novel abstraction that captures their performance characteristics.

3.6 Limitations and discussion

In this section, we highlight some of HammerSlide's and SlideSide's limitations and discuss how they can be addressed as part of future work.

Time-based windows. For SlideSide, we assume incremental computation over partial aggregates, simplifying the implementation and data structure allocation of the algorithms described in Section 3.4. For example, a window of size 60 and size 1 second requires a circular buffer with 60 slots, as we only store the partial aggregates of every second, not individual tuples. To perform incremental aggregation directly on the input stream for time-based windows, additional logic is required for resizing all data structures if the initial size is insufficient.

Holistic functions. While HammerSlide can accelerate the computation of holistic functions, similar to SlideSide, it will have to recompute the result from scratch after every insertion. Existing approaches incorporate a queue for the input tuples with an order statistics tree [105] or an indexable skiplist [104] to keep track of holistic aggregates in $O(\log(n))$ time. We leave for future work the design of hardware-efficient operators that compute optimally such functions.

Out-of-order data. With respect to the ordering guarantees of the input data, HammerSlide can handle "in-order" or "slightly out-of-order" tuples that can be buffered and reordered. SlideSide can perform incremental aggregation of partial aggregates over "out-of-order" tuples lazily, similar to previous approaches [212]. For low latency use cases, in which buffering data is prohibitive, both algorithms could employ punctuation tuples [32] or low watermarks [6] to sort tuples deterministically within a stream.

Workload adaptivity. As discussed in Section 3.5.5, our results reveal that current window aggregation techniques do not exhibit robust performance across different aggregation functions and concurrency levels. Thus, an SPE will perform poorly for different points within this design space or maintain multiple algorithms with a cost model. In Chapter 4, we address this demand by introducing an abstraction that generalizes the approaches above and generates code based on the workload characteristics.

Applicability. HammerSlide can be integrated into Java-based SPEs, such as Apache Flink [45] or Spark Streaming [231], by utilizing the Java Native Interface (JNI) calls

and the Java NIO Direct Buffers, as we demonstrated in Section 3.5.3. However, to exploit HammerSlide performance benefits, it is crucial to manually manage memory outside of the JVM (similar to our solution above or as Apache Flink operates) to avoid expensive copying. SlideSide can be used as a drop-in replacement for any associative aggregation operator in a commercial streaming system, such as Flink [45] (e.g., as an aggregate store for Scotty [212]).

3.7 Summary

In this chapter, we studied the efficient implementation of single-core window aggregation. We first conducted an in-depth study of the best performing algorithms for sliding and tumbling windows and concluded that incremental algorithms are highly CPU-inefficient while work efficient. To address that problem, we developed HammerSlide, a set of optimization techniques, and applied them to the two fastest incremental aggregation algorithms, i.e., SoE and TwoStacks. The result is highly CPU-efficient incremental algorithms that perform parallel processing within a partial aggregate. In fact, they are competitive with non-incremental algorithms for tumbling windows and up to 80% faster for sliding windows. This prevents the need for non-incremental streaming window aggregation and, thus, holds the potential to not only make SPEs faster but simpler as well.

In the second part of this chapter, we presented a novel algorithm for the highly efficient sharing between multiple aggregate queries, called SlideSide. SlideSide complements the SIMD-parallel processing of HammerSlide by performing incremental computation with a *prefix-* and a *suffix-scan* over the partial aggregates generated. SlideSide outperforms the state-of-the-art algorithms in multi-query scenarios by up to 2× in throughput while exhibiting better latency.

We consider the presented work the first step towards a highly hardware-conscious SPE. Naturally, many components of that system are still missing, such as a complete set of efficient streaming operators on modern scale-up architectures using *just-in-time* code generation. In Chapter 4, we introduce a novel system design to tackle this challenge.

Chapter 4

Scaling Window Operators on Multi-Core Processors

In the previous chapter, we described a family of techniques that provide CPU-efficient execution and reduce repeated computations for window aggregation. While these techniques accelerate single-core execution, scaling window aggregation to multiple processing units (i.e., multi-core or distributed parallelism) is challenging (see Section 2.4.2). This results in SPEs incorporating ad-hoc execution strategies with suboptimal performance for different queries based on the workload characteristics (e.g., window semantics).

However, given the ever-growing amount of data [186] acquired by multiple sources (e.g., smart home sensors, industrial appliances, and scientific facilities), and the importance of this class of applications in many domains [197, 166, 95, 49, 6], modern SPEs must be designed precisely for the efficient parallel execution of many window aggregation queries. In particular, as processing throughput is a key performance metric in such use cases, SPEs must exploit the multi-core parallelism of modern CPUs [158, 236].

This chapter focuses on the system aspects of a modern scale-up SPE and, more specifically, the analysis, design, and implementation of a hardware-efficient engine that exploits multi-core CPU architectures for window aggregation queries. While many approaches exist for parallelizing stream processing operators [128, 94, 33, 68], SPEs have not yet addressed the scalability challenge of window aggregation on modern CPUs. The first challenge faced by SPEs is the resolution of the trade-off between parallelism and incremental execution required to increase resource utilization. The second challenge is the need to preserve the ordering semantics of window operations while allowing concurrent state access (i.e., expiring and updating windows). Finally, to enable multi-core scaling, SPEs must account for memory hierarchies and address the overhead of the NUMA effect.

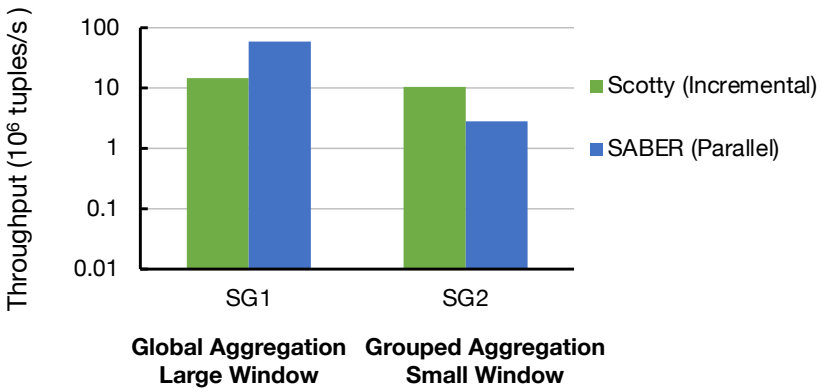


Fig. 4.1 Need to balance incremental and parallel execution

As discussed in Section 2.3, 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 [59, 184, 193]. In contrast, sliding windows offer a new design challenge, which has not been comprehensively explored, especially on multi-core execution. When answering a query with sliding window aggregation, we observe a tension between techniques that use (i) *task-* and *data-level parallelism*; and (ii) *incremental processing*, avoiding redundant computation across overlapping windows and queries. Parallel execution provides CPU efficiency but requires synchronization primitives for concurrent window operations and hardware-conscious designs (i.e., utilize memory hierarchies). At the same time, incremental processing introduces control and data dependencies among CPU instructions, reducing opportunities for exploiting parallelism (see Section 3.2). Therefore, existing engines implement ad-hoc aggregation and parallelization strategies that achieve high performance for specific queries.

To demonstrate this problem, in Figure 4.1, we compare SABER against Flink using Scotty [1] for two different queries over event traces generated by a smart electricity grid [123]: (i) a query that computes a global aggregate with a large sliding window (SG_1); and (ii) a query that computes a grouped aggregate with a smaller sliding window (SG_2).[2] These two systems represent two distinct approaches: one that performs efficient parallelization and one that performs efficient incremental execution. For the first query, we observe that the efficiency coming from incremental execution is not enough, while for the second, the parallel system now does not yield the best performance. As a result, we conclude that we need to consider both for designing a general solution.

When focusing on how SPEs implement incremental execution, we observe no dominating approach. In particular, in Sections 2.3.6 and 3.5.4, we demonstrated experimentally

[1] See Section 2.4.2 for details on their execution strategies.
[2] The queries $SG_{1\text{-}2}$ are denoted with the CQL Syntax in Section A.1.

that existing SPE designs [15, 211, 139, 24] pick a point in the trade-off space of window aggregation approaches. Consequently, they do not exhibit robust performance across query types. For example, as Figure 2.12 shows, when comparing four state-of-the-art approaches [106, 194, 200, 4, 207], each approach outperforms the others for some part of the workload but is suboptimal in others. Some approaches exploit the invertibility property [106, 194] to increase performance; others [207, 200] efficiently share aggregates for non-invertible functions. We conclude that *a modern SPE should provide a general evaluation technique for window aggregation queries that always achieves the best performance irrespective of the query details.*

Opportunity. In this chapter, we argue that with highly-parallel heterogeneous architectures and modern network technologies (Section 2.2.3) becoming a commodity in data centers today, there is a unique opportunity for designing SPEs that exploit previously unseen levels of parallelism. We identify such novel scale-up designs as a practical alternative to expensive scale-out approaches. In addition, they avoid end-to-end latencies prohibitive for a set of time-critical applications. For such an SPE design, we target the following mismatches: (i) revisit the trade-off between parallel and incremental execution for window aggregation; (ii) enable parallelization while preserving precise window semantics; and (iii) design operators that utilize hardware resources efficiently.

Requirements. To design a *general-purpose* relational SPE that can transparently take advantage of existing hardware, our system must meet the following requirements (**R1**-**R3**):

1. *Generalize* the *pallalel window aggregation strategies* of existing SPEs. It is crucial to balance parallelism and incremental execution to increase the execution efficiency for all window aggregation queries without compromising window semantics.

2. *Generalize* the *incremetal strategies* of the state-of-the-art algorithms to exhibit robust performance across different query types. An efficient SPE must achieve high performance irrespective of the window definition and the aggregation function type of the executed queries.

3. Ensure *high throughput* and *low-latency* results despite the workload characteristics (i.e., number of distinct keys or aggregation function). For continuous queries to make progress quickly, SPEs have to guarantee *high resource utilization*.

In this chapter, we describe LIGHTSABER, a novel scale-up SPE that balances parallelism and incremental processing and uses JIT query compilation for the efficient execution windowed stream queries. We start with an overview of LIGHTSABER and its components in Section 4.1. In Section 4.2, we introduce a general model that not only captures

window aggregation strategies found in existing SPEs but also allows us to define new ones. Subsequently, in Section 4.3, we describe LIGHTSABER's approach for parallel aggregation (**R1**) using a tree abstraction, called a *parallel aggregation tree* (PAT). We then describe how LIGHTSABER generates code for stream queries (Section 4.4) and focus on efficient incremental code generation (**R2**) based on the *generalized aggregation graph* abstraction (Section 4.5). In Section 4.6, we discuss LIGHTSABER's implementation details (**R3**), followed by our experimental evaluation (Section 4.7), the limitations of our system (Section 4.8), and conclusions (Section 4.9).

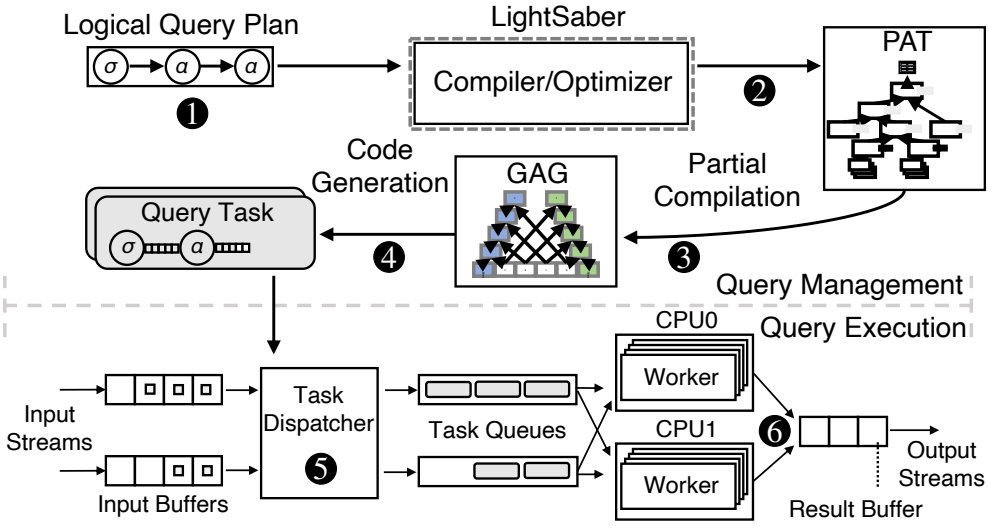


Fig. 4.2 LIGHTSABER system design

4.1 Overview

With the elimination of interpretation overhead due to the recent trend to implement query engines as code generators [165, 191], the differences in the window evaluation approaches have a more pronounced effect on performance. Even though generating executable code from a relational query is non-trivial (as indicated by the many papers on the matter [176, 165, 191]), it is fundamentally a solved problem: most approaches implement a variant of the compilation algorithm by Neumann [165]. 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 mentioned above, as stated in R1 and R2 requirements.

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 [188, 176]. This chapter aims to develop just such a new abstraction for evaluating window aggregation queries. We do so by dividing the problem into two parts: (i) effective parallelization of the window computation (**R1**) and (ii) efficient incremental execution as part of code generation (**R2**). We develop abstractions for both and demonstrate how to combine them to design a novel scale-up SPE, LIGHTSABER.

At a high-level, users submit queries to LIGHTSABER that get translated into logical operator graphs. These graphs are optimized with logical optimizations before window aggregation operators are expanded to *parallel aggregation trees*, which manage parallel execution. From partially evaluating the *parallel aggregation trees*, aggregation graphs are constructed for efficient incremental execution. After these steps, LIGHTSABER generates code for different operators. Finally, as new data arrives, it creates and schedules new tasks for workers based on data locality. Next, we discuss the key components of our design shown in Figure 4.2.

Imperative API. LIGHTSABER allows users to express relational stream window queries [21] with the operators discussed in Section 4.4.1. To illustrate a simple streaming operator graph, in Figure 4.3, we implement an application that filters an input stream with tumbling windows using LIGHTSABER's imperative API.

Example: In the application of Figure 4.3, we employ the `SELECT` operation with a tumbling count-based window (i.e., `ROW_BASED`). In lines 1-2, a selection is defined with a predicate expression that checks if the first column of an input tuple is equal to 3. An operator is constructed (lines 4-7), and multiple operators can be connected to create a dataflow execution graph, which we refer to as `Query` in line 8. Next, multiple execution graphs can be combined, resulting in an `QueryApplication` that ingests and analyzes data continuously (e.g., from a TCP socket) in lines 12-13. The *parallelism degree* is set by the number of available processing units (i.e., `NUM_OF_WORKERS`).

Compiler. LIGHTSABER uses a compiler to perform JIT query compilation based on the following steps: (i) it parses and optimizes the execution graph defined by the imperative API using a set of logical rules [218, 21, 107] in stage ❶; (ii) it translates the execution graph into a *parallel aggregation tree* (PAT) that defines the execution strategy of parallel processing in stage ❷; (iii) it partially evaluates the tree into an aggregation graph used for incremental execution in stage ❸; and (iv) it code-generates an incremental algorithm from the aggregation graph in stage ❹. Let us now analyze the individual steps of the compiler by introducing our general window aggregation model and then discussing the abstractions for parallel and incremental aggregation (requirements **R1** and **R2** respectively).

```
1  auto predicate = new ComparisonPredicate(EQUAL_OP, new ↩
   ColumnReference(1), new IntConstant(3));
2  auto selection = new Selection(predicate);
3  auto window = new WindowDefinition(ROW_BASED, 60, 60);
4  auto cpuCode = new OperatorKernel()
5                          .setInputSchema(getSchema())
6                          .setSelection(selection)
7                          .setup();
8  auto query = new Query(new QueryOperator(*cpuCode), *window);
9  auto application = new QueryApplication(query)
10                         .setup(NUM_OF_WORKERS);
11 while (true) {
12   input = ...;  /* get a pointer to a batch of data */
13   application->processData(input);
14 }
```

Fig. 4.3 Imperative API for LIGHTSABER (`SELECTION` operator example)

In Section 4.2, we formalize window aggregation strategies as part of a general model that allows us to express approaches found in existing systems and 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 data-level (i.e., SIMD) and task-level (i.e., multi-core) parallelism while retaining a degree of incremental processing.

For the parallel computation of window aggregates ❷, in Section 4.3, 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 next level to produce sequences of window fragment results, which are finally combined into a stream of window results.

To generate executable code from nodes in the PAT ❸, in Section 4.5, 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.

Runtime. LIGHTSABER provides a runtime that manages query execution on multi-core CPUs (requirement **R3**). It uses a centralized task scheduler for each query pipeline [165] that creates tasks when new data arrives at ❺ using the parallelization strategy defined by

the *PAT*, as discussed in Section 4.4. These tasks are placed in system-wide queues based on their NUMA-locality, and workers execute them and place the partial or complete results in output queues in stage ❻. During this stage, the workers attempt to perform the last level of PAT, i.e., merge window fragments in an out-of-order manner while respecting window semantics. To execute long-running applications efficiently, LIGHTSABER manages its own memory and utilizes all available resources using NUMA-aware scheduling (see Section 4.6).

LIGHTSABER engine. Based on the above components, we design and implement LIGHTSABER, an SPE that balances parallelism and incremental processing on multi-core CPUs. Even though our main contributions tackle the challenge of parallel window aggregation, our design also addresses inherent problems of efficient multi-core parallelization, such as utilizing cache hierarchies and scaling under the NUMA effect. We, thus, provide a general approach for accelerating streaming SQL queries [21] with relational windowed operators (e.g., `JOINs`, `AGGREGATIONs`, or `SELECTIONs`) for both single and multi-query applications.

Our 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 [58], and up to one order of magnitude for other queries. We confirm that GAGs generate code that achieves high throughput with low latency compared to existing incremental approaches through micro-benchmarks. On a 16-core server, LIGHTSABER processes tuples at 58 GB/s (470 million tuples/s) with 132 μs latency.

4.2 Modeling window aggregation strategies

As discussed in Section 2.3.6, we can exploit the properties of the aggregation functions to compute aggregates either in parallel or incrementally. Surprisingly, we found that current stream processing designs do not take advantage of this. Evidently, there is a design space for SPEs without a dominating strategy for window aggregation. Therefore, what is lacking is a model that captures the design space and allows reasoning about design decisions and their impact on system performance.

We formalize different aggregation strategies as part of a general 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*). As described in Section 2.3.2, a pane is a non-overlapping partial aggregate and a sash is a window fragment that consists of multiple panes. The DFA in Figure 4.4 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 [46]), which can be merged and aggregated into sashes in one or more (i.e., hierarchical) steps. Sashes are combined into

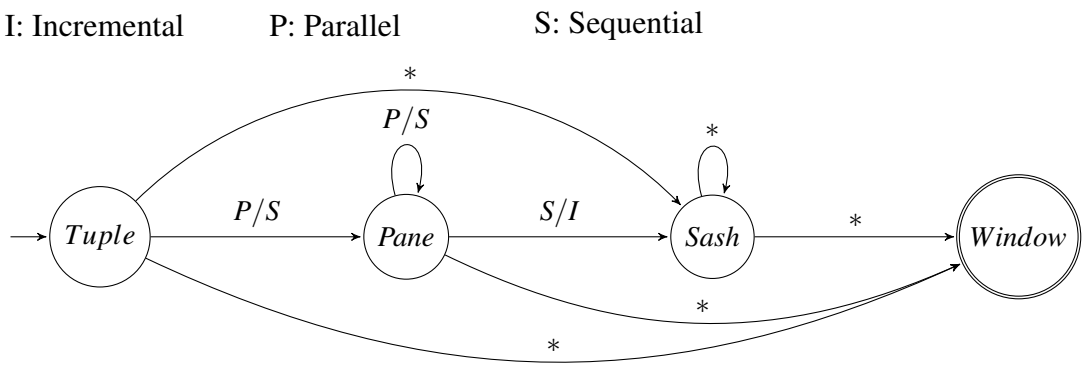


Fig. 4.4 Model of window aggregation design space (* is the Kleene-star)

complete windows (also, potentially hierarchically). Conceptually, each of those aggregation steps can be *sequential*, *parallel* or *incremental*.

Given this model, a system is described by a word of the form $(S \rightarrow Pane, PPI \rightarrow Sash, I \rightarrow Window)$.[3] 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 using the SPEs from Table 2.2. Systems that utilize the bucket-per-window approach, such as Flink, Spark, StreamBox or BriskStream, have an aggregation tree that has only one (incremental) level ($I \rightarrow Window$) in our model. Slicing techniques atop Flink, such as Cutty [46] and Scotty [212], remove redundant computations and are either $(S \rightarrow Pane, I \rightarrow Window)$ or $(S \rightarrow Pane, S \rightarrow Window)$.

In Section 2.4.2, we discussed how SABER decomposes its operator execution into: a *fragment function*, which processes a sequence of window fragments and produces *sashes*; and an *assembly function* that constructs complete results from the *sashes* and reorders them based on the window semantics. Therefore, it has three processing states for its CPU execution model: it incrementally aggregates tuples into sashes, aggregates multiple sashes in parallel but generates complete windows sequentially. Formally, SABER implements $(I \rightarrow Sash, PS \rightarrow Window)$.

While SABER is arguably the most advanced of these systems, it 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. The following section describes how LIGHTSABER overcomes these limitations by occupying a new point in this design space.

[3] Note that spaces, commas, and $\rightarrow$ are merely for readability and do not have formal semantics.

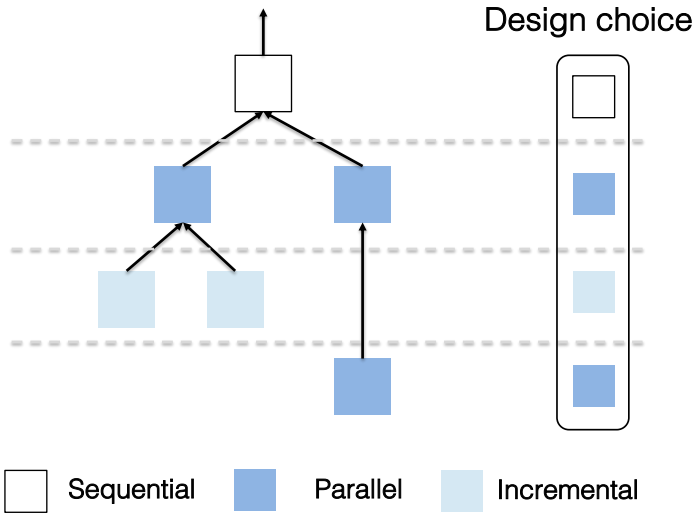


Fig. 4.5 Defining a window aggregation strategy

4.3 Parallelizing window aggregation with trees

Having formalized the design space, let us now describe how to select an appropriate physical evaluation strategy. Figure 4.5 illustrates an aggregation scenario in which the boxes represent window computations: all of them could be parallel, incremental, or sequential. If these computations were randomly assigned a different execution strategy, the interpretation overhead would complicate the code-generation process. Therefore, we limit the design space by selecting a common computation approach for every level.

The execution strategy now is a slice through the tree of Figure 4.5, and we have a design space where the optimal strategy is highly query-, input- and system-specific. In particular, the aggregation step from sashes to windows can be performed hierarchically (i.e., following the self-referencing loop at the sash-node in Figure 4.4 multiple times). The optimal number of hierarchical merge steps is challenging 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.

However, implementing such a design is non-trivial because it combines static sub-plans (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)

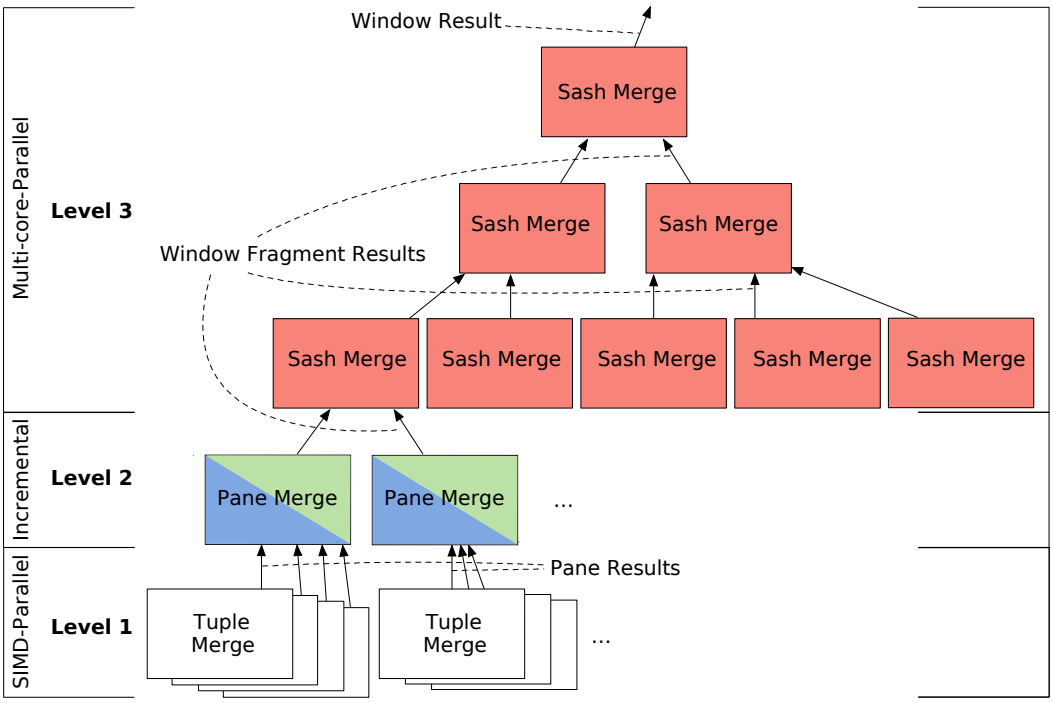


Fig. 4.6 Parallel aggregation tree in LIGHTSABER: (i) the green and blue colors indicate the computation of prefix- and suffix-scan by GAG (see Section 4.5); (ii) the red color indicates the computation of higher level aggregates.

(see Figure 4.6). A PAT has three distinct levels 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, allowing LIGHTSABER to scale to more parallel hardware for queries that do not have 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.

4.3.1 Multi-level parallel window aggregation

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 downsides. LIGHTSABER partitions the input stream into panes similar to the strategies pioneered by Pairs [140] or Cutty [46]. Within each pane, LIGHTSABER exploits data-level parallelism in line with HammerSlide optimizations (see Section 3.3).

The panes are computed using a specialized data structure, which we call a *pane aggregation table* (see Figure 4.7). The *pane aggregation table* is shared between the two

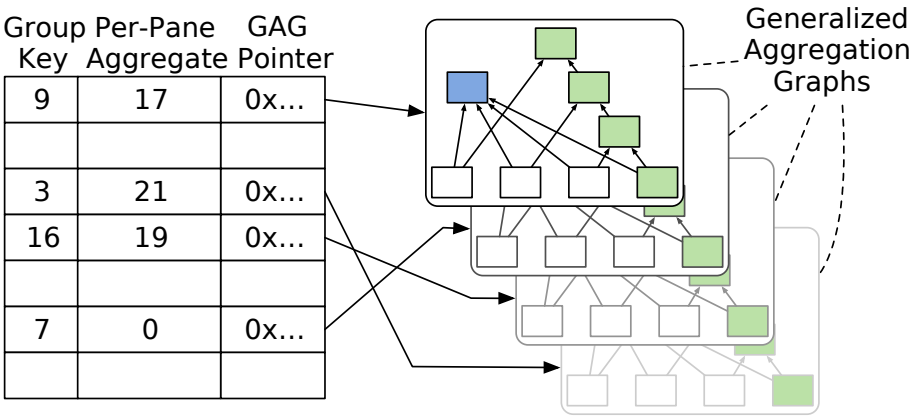


Fig. 4.7 Pane aggregation table

bottom levels of Figure 4.6. 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 Section 4.5. GAGs are abstract because 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. GAGs support the insertion of new elements with an interface similar to a queue, 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, marking the time when it no longer contributes to windows and is safe to remove it.

While evaluating 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 to generate code executed as a single, fully-inlined task at runtime. This reduces the number of function calls and improves data and instruction locality. However, SIMD parallelization for grouped aggregations 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. While the previous two levels operate on disjoint sets, the sash merge requires coordination for preserving precise window semantics: sashes from multiple individual tasks may contribute to the same complete window result.

At first, each window fragment is tagged as opening, closing, complete, or pending, and an increasing logical identifier (i.e., window id) is assigned to it, similar to SABER [137]. 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 to ensure ordering guarantees. 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. Finally, complete windows are sent downstream ordered by their window ids.

4.3.2 Discussion

While we have identified a $(P \rightarrow Pane, I \rightarrow Sash, P^*S \rightarrow Window)$ strategy as optimal for window aggregation on multi-core CPUs, our generalized model also allows us to express novel strategies for distributed or heterogeneous architectures [137, 208]. For example, when executing streaming applications with ungrouped aggregations on GPGPUs, previous work has shown [137] that it is more beneficial to employ a $(P \rightarrow Window)$ design. We want to investigate the optimal aggregation strategies for more SPE designs in future work.

4.4 Generating code for operators at the query level

LIGHTSABER generates code for two different purposes: at the query level, code is generated to implement the operator semantics (selections, 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. As discussed in the previous section, the pane aggregation table connects these two processing levels. In this section, we first introduce the query-level code generation before presenting our code generation approach for incremental aggregation in Section 4.5.

4.4.1 Supported relational 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 data 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, avg); and (iii) the `JOIN` (⋈) operator, which allows joining a stream with another stream using `WINDOWS` or a static table. All operators are deterministic and expect "in-order" input streams (see temporal ordering in Section 2.1.2). Users can extend LIGHTSABER with new deterministic operators by implementing a *fragment function* and an *assembly function*, as discussed in [137].

`GROUP-BY` and `JOIN` with a static table are implemented using generated (and, thus, inlined) hashtables as well as aggregation code. Following common practice, we use Intel's SSE CRC instruction [131] and bit masking for hashing, open addressing for hashtable layout, and linear probing for conflict resolution. The hashtables are pre-allocated according to (generous) cardinality estimates but can be resized (and rehashed on overflow).

`WINDOW JOIN` between two streams is implemented as a streaming θ-join [126]. Similar to SABER, LIGHTSABER achieves parallelism by the data-parallel execution of tasks instead of utilizing a task-parallel implementation [89].

For each query pipeline [165], a centralized task scheduler: (i) buffers the input stream (or streams from both sides for `JOIN`); (ii) creates tasks over disjoined sets based on window semantics; and (iii) assigns each task monotonically increasing task identifiers. These task identifiers allow LIGHTSABER to reorder complete window results before emitting them downstream. In the case of `JOIN`s, LIGHTSABER discards data from a stream only if they have been joined with all relevant data from the other.

4.4.2 Generating query code using LLVM

To translate the non-incremental operators to executable code, we follow the approach by HyPeR [165]: we traverse the operator tree and append instructions to a code buffer until reaching an operator that requires materialization of its result (i.e., a pipeline breaker). The pipeline breakers for LIGHTSABER are stream-to-stream `JOIN` and `WINDOW AGGREGATION`. Every sequence of pipelineable operators translates to a single execution task for the workers (see stage ❺ in Figure 4.2), which minimizes task overhead (i.e., allocation, result passing, and scheduling). Subsequently, each execution task is decomposed into a *fragment* and an *assembly function* similar to SABER [137], but the *assembly* functions can be performed in parallel as discussed in the previous section. As we show in Figures 4.8b and 4.9b, the main work of the *fragment functions* is performed in a tight loop over the input stream(s), which improves data and instruction locality. Each task is optimized and compiled into executable code by the LLVM compiler framework [142].

To illustrate this process for `WINDOW AGGREGATION`, consider the query in Figure 4.8a (denoted in *continuous query language* syntax [21]). Taken from the Linear Road Benchmark [22], it reports the road segments on a highway lane with an average speed lower than 40

```
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) CQL 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 the 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()

Fig. 4.8 Query code generation for `WINDOW AGGREGATION` in LIGHTSABER

over a sliding window with size 300 and slide 1 second. A simplified version of the generated code for the *fragment function* of this query (in C++ notation) is shown in Figure 4.8b. Note that the LIGHTSABER query compiler combines levels 1 and 2 from Section 4.3, as illustrated in Figure 4.6, 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-by key calculation) but expresses aggregation purely in terms of the pane aggregation table API. The pane aggregation table implementation (Figures 4.8c and 4.8d) 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`. While LIGHTSABER supports only tumbling

```
select L.timestamp, L.plug, L.household, L.house
from LocalLoadStr  [range 1 slide 1] as L,
     GlobalLoadStr [range 1 slide 1] as G
 where L.localAvgLoad > G.globalAvgLoad
```

(a) CQL for the join sub-query of SG_3

```
1  while (index1 < lBatch.size() && index2 < rBatch.size()) {
2    time1 = lBatch.getTimestamp(index1);
3    time2 = rBatch.getTimestamp(index2);
4    /* Process the right window */
5    if (time1 <= time2) { /* simplified windowing condition */
6      lTuple = lBatch[index1];
7      /* scan the right window and evaluate predicate */
8      for (tuple &rt: rightWindow) {
9        if (rt._1 == lTuple._4) { /* ... */}
10     }
11     /* add current tuple to leftWindow */
12     leftWindow.add(lbTuple);
13     /* remove old tuples from leftWindow and rightWindow */
14   } else {
15     /* process symmetrically the left window */
16   }
17 }
```

(b) Generated code for the fragment function of SG_3

Fig. 4.9 Query code generation for `WINDOW JOIN` in LIGHTSABER

and sliding windows, it can be extended to support user-defined windows by generating a more complex condition for window construction in line 3.

For `WINDOW JOIN`, we consider the query SG_3 in Figure 4.9a that detects anomalies in a trace from a smart electricity grid [123] and present a simplified version of the generated *fragment function* in Figure 4.9b. The main loop of the generated code iterates over input batches from the left and right side of the θ-join to create windows similar to the logic from previous work [126]. Before a new tuple is added to a window (line 12), it is joined with all the tuples from the window constructed from the other stream (lines 8-10), and it invalidates old tuples from both windows (line 13). Additional `SELECTION` and `PROJECTION` operators can be applied in line 9.

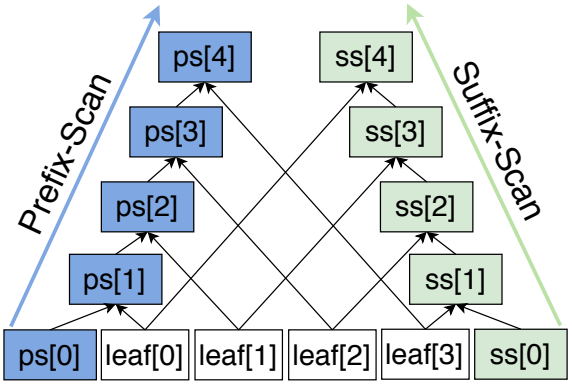


Fig. 4.10 Initial Generalized Aggregation Graph

4.5 Generating code for aggregation strategies with Generalized Aggregation Graphs

In the previous section, we discussed generating code at a query level (i.e., operator semantics). Next, we introduce our code generation approach for incremental aggregation based on an abstraction called a *generalized aggregation graph* (GAG). The objective of GAGs is to combine the benefits of code generation (hardware-conscious, function-call-free code) with those of efficient incremental processing. It allows us to capture the design space of the best (known) in-order incremental processing algorithms described in the previous sections and generate optimized code. This is implemented by instantiating the appropriate query fragments (i.e., "templates") at runtime.

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 that connects them with the pane aggregation table (Section 4.5.1), the creation of an initial generic GAG that captures different strategies (Section 4.5.2), the specialization to a specific aggregation strategy (Section 4.5.3), and the translation into executable code (Section 4.5.4). Finally, we discuss the optimizations related to multi-query processing (Section 4.5.5).

4.5.1 Programming interface for GAGs

As discussed in Section 4.4.2, the generated code of GAGs relies on three functions to enable efficient shared aggregation within a pane aggregation table:[4]

- `void insert (Value v)`: inserts an item of type `Value` in the GAG and performs any internal changes necessary to accommodate further operations.

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

- `void evict ()`: removes the oldest value and perform the necessary internal changes.
- `Value query (size_t windowSize)`: returns the result with respect to the current state and a given window size.

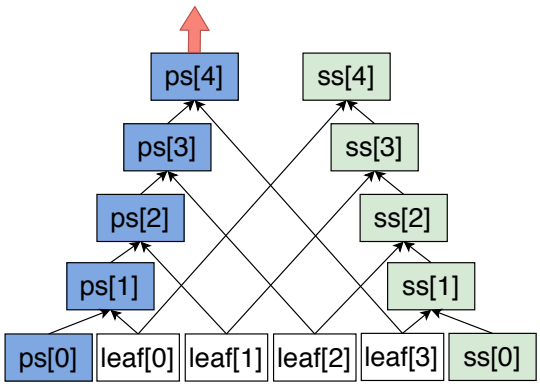
While the first two are rather obvious, the `query` function is interesting in that it indicates that GAG produces results for different window sizes. Such inter-query sharing requires maintaining partial aggregates in memory and efficiently supporting in-order range queries. LIGHTSABER generates shared partials (see Figure 4.8b), and GAGs take care of storing them and producing window results by invoking the `query` function.

4.5.2 Capturing incremental algorithms' dependencies with a graph

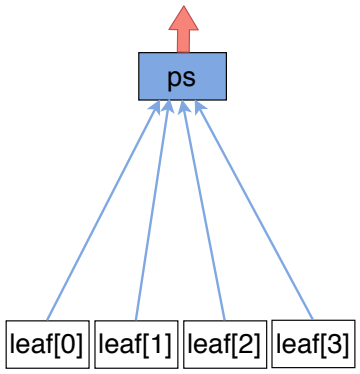
Since each of the presented algorithms in Table 2.1 is the best-performing in parts of the problem space, a GAG must capture their behavior. Subsequently, based on the workload characteristics (window semantics, aggregation function, and number of queries), a GAG must be translated into a dataflow graph that exhibits the same performance characteristics as SoE, TwoStacks, SlickDeque, or SlideSide. At first glance, this seems to be a challenging problem that requires a complex solution. However, the observation that any associative aggregation function can be represented as a binary combination of an element of a *prefix-scan* and a *suffix-scan* of the input simplifies the problem (see definitions in Section 2.3.3). This representation is sufficient to capture the low-level dependencies of the aforementioned incremental algorithms.

Tangwongsan et al. [202] make a similar observation when developing FlatFAT, a binary aggregation tree (see Section 2.3.4). 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 [106]. This results in a dataflow graph as the one shown in Figure 4.10, which can represent any associative aggregation. As indicated by the color-coded nodes (which mirror those of Figure 2.10), 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 of Figure 4.10, the green *suffix-scan* parent nodes (abbreviated as `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 (see Section 2.3.5). These nodes' values can be updated either lazily (upon calling the `query` function) or eagerly (upon tuple `insertion`). Next, we describe how we exploit this property to specialize the initial GAG into one of the previous aggregation algorithms.



(a) Initial inv-GAG



(b) Specialized inv-GAG

```
1 result = 0;
2 for (/* iterate i over input */) {
3   ps = ps + input[i] - input[i-WINDOW_SIZE];
4   /* emit result */
5   result = ps;
6 }
```

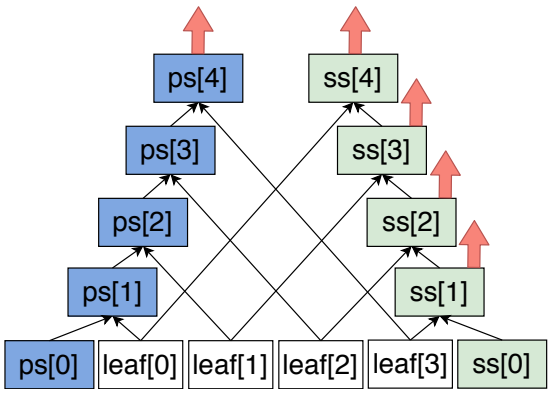
(c) Generated code for invertible functions (sum)

Fig. 4.11 GAGs with invertible functions

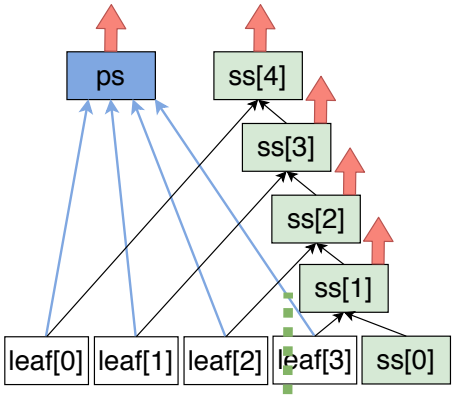
4.5.3 Specializing GAGs depending on the workload characteristics

A GAG must turn a workload specification (i.e., window and aggregation function type) into specialized generated code. The first step in GAG specialization is the removal of unnecessary nodes, which would lead to "dead code". This removal is performed as a simple "mark and sweep" optimization: every node needed to produce final aggregates is marked as required. Afterward, 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 defining a lazy or eager dataflow computation strategy based on the rationale that follows. A *prefix-scan* can be efficiently calculated without buffering (i.e., eagerly) because it only requires access to the last computed element and the current one. At the same time, a suffix-scan requires buffering because it needs to access older tuples from the input stream. In addition, a suffix-scan must scan the whole window, which incurs a linear cost. It is, therefore, beneficial to perform the suffix-calculation lazily whenever tuples are evicted from the window. We refer to the respective tuple ingestions as a "trigger point" and represent it with a dashed vertical line in Figure 4.12b. Note that the suffix-scan directly corresponds to the "stack-flipping" of the TwoStacks algorithm, and it is required to guarantee correctness, as discussed in Chapter 3.



(a) Initial non-inv-GAG



(b) Specialized non-inv-GAG

```
1  result = INT_MAX;
2  ss[WINDOW_SIZE+1] = {INT_MAX};
3  ps = INT_MAX;
4  leafIterator = 0;
5  for (/* iterate i over input */) {
6    /* compute a suffix-scan over the current values */
7    if (leafIterator == WINDOW_SIZE) {
8        for (j = 0; j < WINDOW_SIZE; j++)
9            ss[j+1] = min(ss[j], input[i-WINDOW_SIZE+j]);
10       leafIterator = 0;
11   }
12   ps = min(ps, input[i]);
13   /* emit result */
14   result = min(ps, ss[WINDOW_SIZE-leafIterator]);
15   leafIterator++;
16 }
```

(c) Generated code for non-inv functions (min)

Fig. 4.12 GAGs with non-invertible functions

To illustrate the expressive power of this approach, let use describe the two most illustrative cases: single-query aggregation using an invertible (Figure 4.11b) and a non-invertible (Figure 4.12b) combiner function. Note that we draw *eager* computation edges in blue and *lazy* edges in black.

Figure 4.11a 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 be eliminated. After removing all unmarked nodes and replacing the parent edges with their children, only a single node remains (Figure 4.11b). This representation corresponds to the dataflow graph of the SoE algorithm.

In the case of a non-invertible function (Figure 4.12a), 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 Figure 4.12b. 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.

4.5.4 Generating code for single-window processing

In the previous section, we briefly presented the generated code for two separate scenarios of single-window processing. We now provide a more extensive discussion starting with the simplest case: computing an invertible function. The code (displayed in Figure 4.11c) 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).

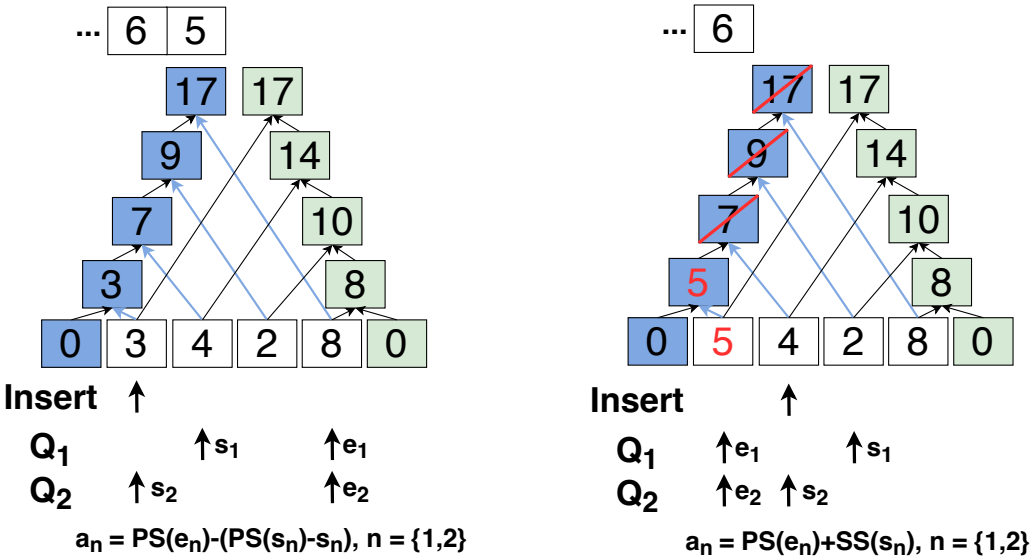
The case of a single-window computing a non-invertible function is more complicated. Conceptually, it corresponds to the TwoStacks algorithm (in particular its implementation in HammerSlide from Chapter 3): as illustrated in Figure 4.12c, the implementation maintains a single aggregate value for the 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.

When computing multiple aggregation functions that belong to both categories (i.e., invertible and non-invertible) over the same window, the specialized graph generated from the GAG is a combination of Figures 4.11b and 4.12b. For the generated code, instead of maintaining a single `ps` back stack value, it now has a separate variable for each aggregation.

4.5.5 Generating code for multi-window processing

The naïve approach to evaluate multiple concurrent window queries with the same aggregation functions would be to have several instances of the single-window code. Instead, we find that we can answer multiple window 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 the input tuples are stored in a circular buffer, which leads to the `start` and `end` cursors of a window changing their relative order while processing the stream. This case is referred to as the inverted buffer problem [202].

To illustrate this problem, consider the example in Figure 4.13a: it shows the evaluation of two queries, Q_1 and Q_2, both calculating an invertible window sum but with different window sizes (3 and 4 elements, respectively). Figure 4.13a shows the state at time $t0$, when Q_1's window contains the values $\{4,2,8\}$. Their sum (i.e., 14) can be calculated by



(a) GAG at time t0

(b) GAG at time t1

Fig. 4.13 Example of multi-window processing

subtracting the exclusive *prefix-scan* of the window `start` cursor, $\mathrm{PS}(s_1) - s_1$, from the inclusive *prefix-scan* of the `end`, $\mathrm{PS}(e_1)$. Similarly, the result of Q_2 at time $t0$ is calculated as 17. When transitioning to time $t1$, 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-scan* of the `end`, $\mathrm{PS}(e)$, and the *suffix-sum* of the `start`, $\mathrm{SS}(s)$.

The case of multiple non-invertible functions generalizes the single-window scenario, using only the root of the back stack array `ps`. The key difference between the two workloads 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, which results in a more efficient update process. We discuss in more detail how to overlay these front stacks over the same memory address space in Section 3.4.2.

4.6 Implementation

In this section, we describe how our abstractions for parallel and incremental aggregation can be realized in a system implementation of a multi-core engine called LIGHTSABER.[5] We implement LIGHTSABER in 24K lines of C++14 and use Intel TBB [117] for concurrent

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

queues and page-aligned memory allocation. We focus on the code generation process of LIGHTSABER (Section 4.6.1) and its runtime components (Sections 4.6.2 and 4.6.3).

4.6.1 Code generation

As discussed in Section 4.1, LIGHTSABER uses a compiler to parse an imperative program and translate it to *query* tasks ready for execution. First, using its imperative API, LIGHTSABER generates a logical execution plan that is optimized based on existing techniques [218, 21, 107]. Our current implementation includes the following optimizations: it (i) reorders operators according to selectivity and moves more selective operators upstream [218] (e.g., for selection-window commutativity [21]); (ii) inlines and applies the `HAVING` clause when a complete window result is constructed; and (iii) reduces instructions by removing components related to cross-process and network communication, that introduce conditional branches [236]. Subsequently, the logical plan is translated into a *PAT* and partially evaluated using *GAG*s to generate incremental code. After this step, LIGHTSABER instantiates the appropriate query fragments in C. Using the Clang front-end, it parses these fragments to generate LLVM bitcode, which gets optimized by multiple LLVM transformation passes. The generated code for different *query* tasks can be executed by any processing unit (i.e., worker).

4.6.2 Memory management

As we demonstrate in Section 4.7, the performance of LIGHTSABER is restricted mainly by memory accesses and, hence, it performs its own memory management for improved performance. The processing of window operators requires the allocation of memory chunks for storing intermediate window fragments and complete window results. However, dynamic memory allocation on the critical path is costly and reduces overall throughput.

We observe that, for fixed-sized windows, the amount of memory required over execution 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 initializing and maintaining a pool of data structures and intermediate buffers per operator pipeline. For example, each grouped `AGGREGATION` operator manages its own pool of hashtables. Finally, LIGHTSABER also uses statically allocated pools of objects for tasks.

4.6.3 Query execution and NUMA-aware scheduling

A lock-free circular buffer per input stream stores incoming tuples. Each *query* task has a start and end pointer on the circular buffer with read-only access and a window operator function pointer. It also contains a free pointer, indicating which memory has been processed and can be freed in the result stage. During this stage of each operator pipeline, LIGHTSABER assembles and reorders the task results before pushing them downstream at ❻ in Figure 4.2. For efficient result reordering, it uses a lock-free queue with atomics per pipeline while each worker stores task results based on their identifier. In contrast to systems such as Flink and Spark, LIGHTSABER delays window computation until the query execution stage, thus avoiding a sequential dispatching stage that becomes the bottleneck of window operators.

In a multi-socket NUMA environment, high CPU utilization depends on whether tasks are scheduled efficiently to individual worker threads based on data locality [229]. First, LIGHTSABER spills the circular buffers for the input streams between all available CPU sockets. To balance computation, the size of the buffers' fragments varies based on the number of worker threads per NUMA node. For example, instantiating a circular buffer with four slots on two NUMA nodes with the same workers means that: (i) the first two slots are allocated on the first node (using `numa_alloc`); and (ii) the remaining slots are allocated on the second node. In addition to circular buffer spilling, LIGHTSABER employs NUMA-aware scheduling to reduce remote memory accesses. Specifically, it maintains a separate lock-free task queue for each NUMA node. When a new task is created, it is placed in a queue based on locality (i.e., where the circular buffer it points to resides). A worker thread favors tasks from its local queue, and only when there is no more work available does the worker attempt to steal from other nodes to avoid starvation.

4.7 Evaluation

In this section, we evaluate LIGHTSABER and its components to show the benefits of our window aggregation approach over a range of synthetic and real-world datasets described along with our evaluation set-up in Section 4.7.1. Subsequently, we demonstrate that LIGHTSABER achieves higher performance compared to existing solutions in multi-core execution (Section 4.7.2). Then, we explore the CPU-efficiency of the generated code (Section 4.7.3), the scalability and the application latency of LIGHTSABER (Section 4.7.4). In Section 4.7.5, we evaluate the performance benefit of PAT's parallel merge step and LIGHTSABER's memory overhead. Finally, we analyze the efficiency of GAG's generated code against state-of-the-art incremental algorithms (Section 4.7.6).

Table 4.1 Evaluation datasets and workloads

Datasets		**Queries**			
Name	# Attr.	Name	Windows	Operators	Values
Cluster Monitoring (CM) [129, 57]	12	CM_1	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{sum}}$	
		CM_2	$\omega_{60,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$	
Smart Grid (SG) [123]	7	SG_1	$\omega_{3600,1}$	π, α_{avg}	
		SG_2	$\omega_{128,1}$	$\pi, \gamma, \alpha_{\text{avg}}$	
Linear Road Benchmark (LRB) [22]	7	LRB_1	$\omega_{300,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$	
		LRB_2	$\omega_{30,1}$	$\pi, \gamma, \alpha_{\text{count}}$	
Yahoo Streaming (YSB) [58]	7	YSB	$\omega_{10,10}$	$\sigma, \pi, \bowtie_{\text{relation}}, \gamma, \alpha_{\text{count}}$	
Sensor Monitoring (SM) [122]	18	SM_f	*various*	α_f	$f \in \{\text{sum}, \text{min}\}$

4.7.1 Experimental setup and workloads

Hardware. 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`. Unless stated otherwise, all experiments use all cores from the server.

Stream processing engines. We compare LIGHTSABER against both Java-based scale-out SPEs, such as Apache Flink (version 1.8.0) [15], and engines for shared-memory multi-cores, such as StreamBox [158] and SABER [137]. 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 large buffers. We then replay these tuples continuously while updating their timestamps.

Workloads. Table 4.1 summarizes the datasets and the workloads used for our evaluation (see Section A.1 for their CQL definition). The workloads capture a variety of scenarios that are representative of stream processing, while window sizes and slides are measured in seconds if not stated otherwise:

- **Compute cluster monitoring (CM) [220].** This workload emulates a cluster management scenario using a trace of timestamped tuples collected from an 11,000-machine

compute cluster at Google. Each tuple contains metrics about monitoring events, such as task completion or failure, task priority, and CPU utilization. We execute two queries from previous work [137] to express common monitoring tasks [129, 57].

- **Anomaly detection in smart grids (SG) [123]**. This workload performs anomaly detection in a smart electricity grid trace. 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) [22]**. This workload is widely used for the evaluation of stream processing performance [80, 232, 121, 2] and simulates a network of toll roads. We use queries three and four from [137], which correspond to LRB_1 and LRB_2 in this work.

- **Yahoo Streaming Benchmark (YSB) [58]**. This benchmark simulates a real-world 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 [175].

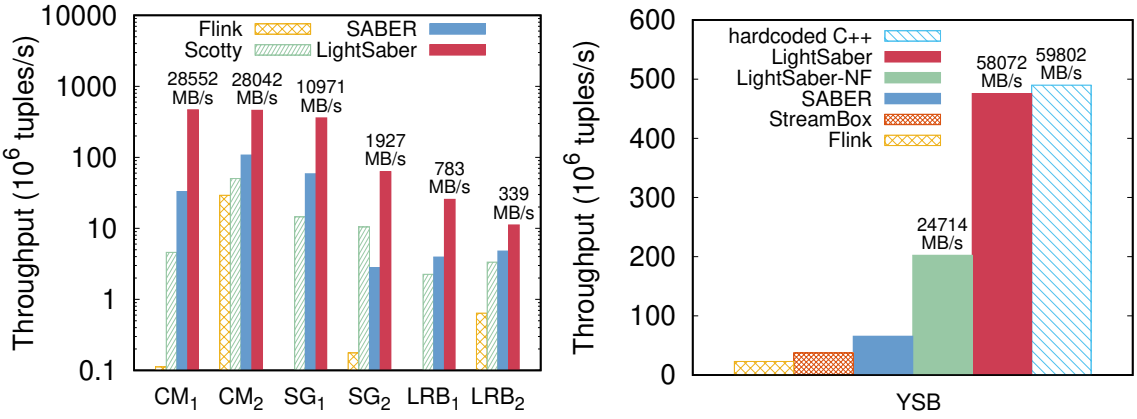
- **Sensor monitoring (SM) [122]**. The final workload emulates a monitoring scenario with an event trace generated by manufacturing equipment sensors. Each tuple is a monitoring event with three energy readings and 54 binary sensor-state transitions sampled at 100 Hz.

Metrics. The main performance metrics considered in the following benchmarks are throughput and end-to-end latency. We define throughput as the average number of tuples processed within a time unit (e.g., one second). We define end-to-end latency as the time between when an event enters the system and when a window result is produced [217]. Candlesticks in plots show the 5^{th}, 25^{th}, 50^{th}, 75^{th} and 95^{th} percentiles, respectively.

4.7.2 Window aggregation performance

To study how LIGHTSABER balances incremental and parallel execution, we use six queries with sliding window semantics from different streaming scenarios and compare its performance against Flink and SABER. Flink represents the bucket-per-window approach [147, 148] that replicates tuples into multiple window buckets. We use the Scotty [212] framework with Flink to provide a representative system with only the slicing optimization.[6] In contrast,

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



(a) Sliding window queries

(b) Yahoo Streaming Benchmark

Fig. 4.14 Performance for application benchmark queries

SABER is a representative example of a system that performs incremental computation on a per-tuple basis and not on partial aggregates.

Figure 4.14a shows that LIGHTSABER significantly outperforms the other systems in all benchmarks. Queries CM_1 and SG_1 have a small number of distinct keys (around 8 for CM_1) or a single key, respectively, which reveals the limitations of systems that parallelize with the *partition-by-key* strategy. Flink's throughput, even with slicing, is at least one order of magnitude lower than that of both SABER and LIGHTSABER. This result shows that current SPE designs do not efficiently support this type of computation out-of-the-box and require explicit load balancing between the operators with custom logic. 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 still has 4×, 9×, and 15× better performance than SABER, Scotty, and Flink, respectively, because it removes redundant computation steps required by the sliding window semantics.

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 due to the expensive hashing operation. 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 step becomes the bottleneck.

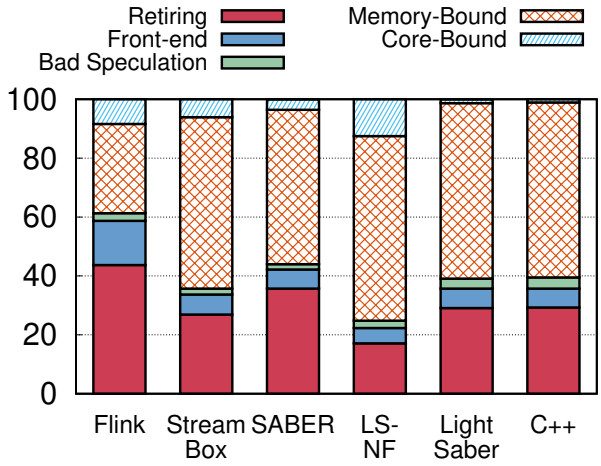


Fig. 4.15 Execution time breakdown

During this step, SABER merges the hashtables from intermediate window results to generate complete windows, which involves the expensive operation of hashing composite group-by keys. 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 optimization from PAT.

Discussion. Overall, we observe that LIGHTSABER outperforms state-of-the-art scale-out SPEs, such as Flink, and efficient scale-up SPEs by at least an order of magnitude and 2× respectively for a range of real-world queries. LIGHTSABER manages to balance parallelism and incremental execution on multi-core CPUs. Therefore, it provides a general and robust design despite the workload characteristics (i.e., number of distinct keys or window semantics) in contrast to existing solutions (see Figure 4.1).

4.7.3 Code generation efficiency

Next, we explore the efficiency of LIGHTSABER's generated code. We use YSB to compare LIGHTSABER to Flink, SABER, StreamBox, LIGHTSABER without operator fusion (denoted as LightSaber-NF or LS-NF), and a hardcoded C++ implementation. StreamBox is a NUMA-aware in-memory SPE with an execution model [143] similar to LIGHTSABER that uses the interpretation-based processing [234]. For this workload, the GAG does not wield performance benefits because there is no potential for intermediate results sharing to exploit. We omit Scotty for the same reason, as slicing does not affect the performance of tumbling windows in Flink. Finally, we conduct a micro-architectural analysis to identify bottlenecks.

As Figure 4.14b 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 (de)serialization, introducing extra function calls and memory copies. We do not observe similar behavior for the other systems, 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 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. We omit the results for the other benchmarks from Section 4.7.2 as we observe similar results when comparing LIGHTSABER with a hardcoded implementation or with an execution strategy without operator fusion.

Figure 4.15 shows a breakdown by CPU components following Intel's optimization guide [116], demonstrating 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. We provide a more detailed description of CPU's hardware components in Section 2.2.1.

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, indicating that they do not efficiently exploit the available CPU resources. 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 due to its interpretation-based model. 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 [234] and copying it between operators; LIGHTSABER and the hardcoded C++ implementation utilize the CPU resources and the memory hierarchy more efficiently. Despite exhibiting better data and instruction locality, they have the highest bad speculation (up to 4%) because slicing and computation are performed in a single tight loop, as shown in Figure 4.8b.

Discussion. Our code generation approach results in highly CPU-efficient code that exhibits the same performance characteristics as optimized handwritten C++ implementations. The SPEs that use an interpretation-based execution strategy (i.e., Flink, StreamBox, and SABER) are core-bound as they fall short of utilizing the CPU resources efficiently. At the same

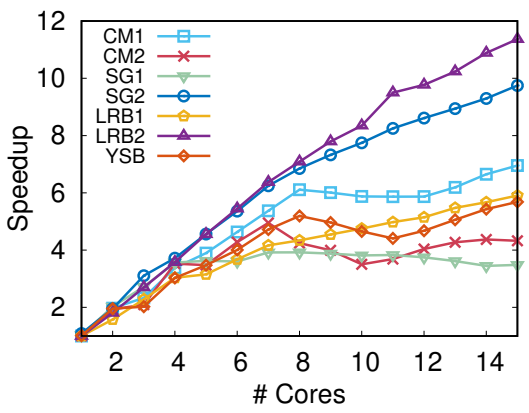


Fig. 4.16 Scalability of LIGHTSABER

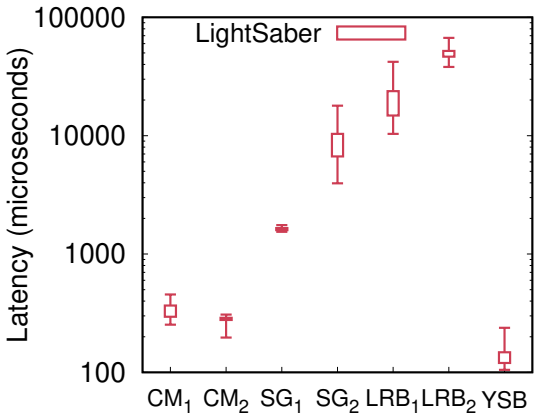


Fig. 4.17 Low end-to-end latency

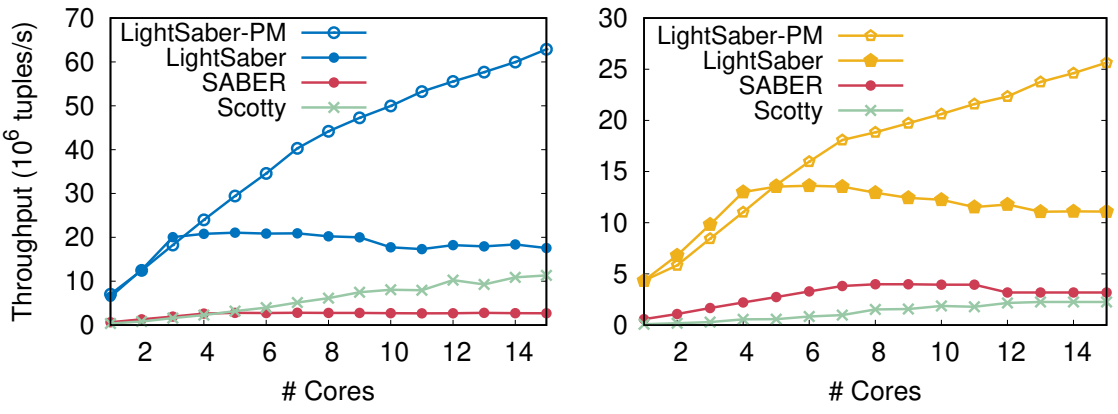
time, the Java-based solutions are more front-end bound and suffer from JVM overheads, such as large instruction footprint or cache misses due to random memory access patterns of allocated objects. On the other hand, our solution induces a higher data and instruction locality, which leads to saturating up to 65% of the memory bandwidth.

4.7.4 Scalability and end-to-end latency

Next, we evaluate the scalability and the end-to-end latency of LIGHTSABER. Starting with the scalability experiments, we use the seven 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, and we, thus, report the results up to 15 worker threads (i.e., we do not use hyperthreading as it reduces the performance due to cache misses).

The results in Figure 4.16 show that LIGHTSABER scales linearly up to seven 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 seven cores, even though the remote memory accesses are kept low. For these four queries, the system is memory bound (up to 60%) and operates close to the memory bandwidth. Therefore, we observe a throughput comparable to 400 million tuples/s and only a 15% performance benefit when LIGHTSABER crosses NUMA sockets. We investigate whether applying software prefetching can yield better performance in Section 5.5.2.

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 because of the expensive hashable probing with composite group-by keys. In this case, the reduction of the remote memory accesses improves the scalability of LIGHTSABER.



(a) SG_2 query (b) LRB_1 query

Fig. 4.18 Parallel merge leads to scalability

Figure 4.17 shows that the median 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 median latency, which is an order of magnitude lower compared to the reported results of other streaming systems [217, 99]. The main reason for this is that LIGHTSABER efficiently combines partial aggregation with incremental computation, which leads to very low latencies.

Discussion. Overall, LIGHTSABER manages to scale to multiple NUMA nodes when evaluating compute-intensive queries. When utilizing another NUMA socket for memory-bound applications, the performance drops slightly or remains comparable to that of seven cores (i.e., first socket only), which reveals that a CPU with increased memory bandwidth could benefit our design. In terms of end-to-end latency, LIGHTSABER exhibits ms results even under high input load. Therefore, it satisfies the strict latency requirements of the *time-critical* applications [235, 166] discussed in Section 2.1.

4.7.5 Measuring parallel merge performance and LIGHTSABER's memory consumption

We now focus on the performance improvement induced by our aggregation approach and its memory requirements, starting from the benefits of PAT's parallel merge phase. In queries SG_2 and LRB_1, aggregations are grouped-by multiple keys with many distinct values, which makes the aggregation expensive, as shown in Section 4.7.2. Probing a large hashtable with many collisions and updating its entries cannot be done efficiently by SABER's single-

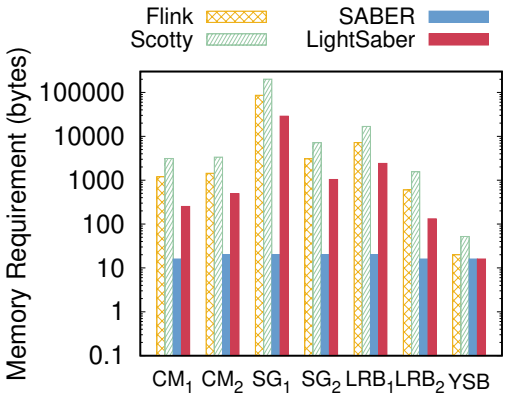


Fig. 4.19 Memory consumption

threaded merge. LIGHTSABER's parallel merge approach removes this bottleneck for such workloads.

In Figure 4.18, we compare the scalability of SABER, Scotty, and LIGHTSABER 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 phase, which results 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 [155].

Figure 4.19 evaluates the memory consumption for the different systems while considering the memory required for storing partial aggregates and metadata as in previous work [212]:

- Flink stores an aggregate and the start/end timestamps per active window.
- Scotty with lazy slicing [212] 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 along and a counter. It uses less metadata compared to Scotty as it operates on deterministic windows.
- SABER directly accesses tuples from the input stream without storing partial aggregates in contrast to the other approaches.

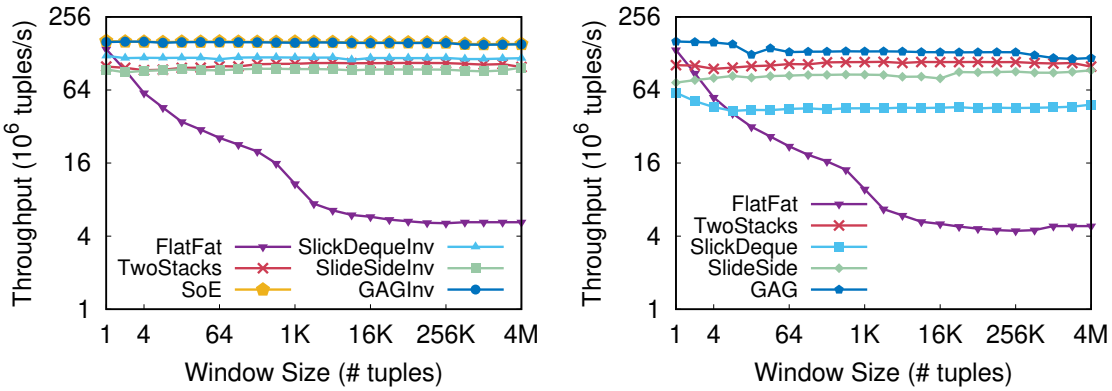
SABER's design exhibits three orders of magnitude lower memory consumption than LIGHTSABER. Without slicing over the input stream, LIGHTSABER can adopt this approach: e.g., apply algorithms from Figures 4.11c and 4.12c on the input stream instead of partial

aggregates. This is more computationally expensive, however, because it requires repeated applications of the `inverse` combiner, leading to worse performance (see Section 4.7.2). Compared with the other approaches, LIGHTSABER requires at least 3× and 7× less memory than Flink and Scotty, respectively.

Discussion. To sum up, PAT's parallel merge phase enables LIGHTSABER to scale the execution of compute-bound window queries. At the same time, its memory requirements are lower than Flink and Scotty while yielding better performance results. In contrast to SABER, though, LIGHTSABER trades off memory consumption for higher throughput.

4.7.6 Evaluation of incremental code generation

In this section, we explore the efficacy of the code generated from GAG for both single- and multi-query workloads using the SM dataset with count-based windows. To evaluate different aggregation algorithms in an isolated environment, we run our experiments as a standalone process to avoid any system interference. Each algorithm maintains sliding windows with a slide of one tuple by performing an eviction, insertion, and result generation, 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 from Chapter 3); (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 (i.e., partial aggregates to be maintained).



(a) Single-query throughput (sum)

(b) Single-query throughput (min)

Fig. 4.20 Comparison of incremental processing techniques for a single-query

Single-query evaluation. For this experiment, query SM_{sum} computes a sum of an energy measure over windows with variable window sizes between 1 and 4 M tuples. As

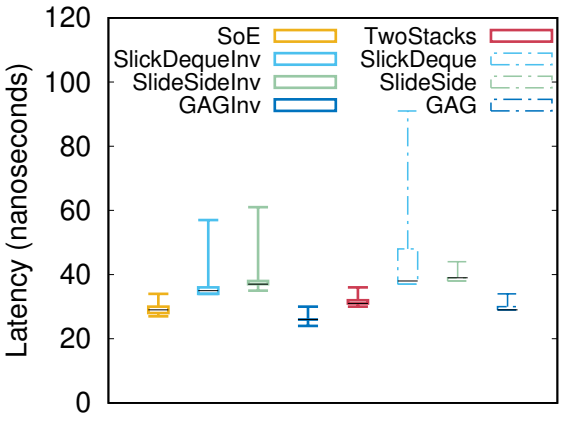
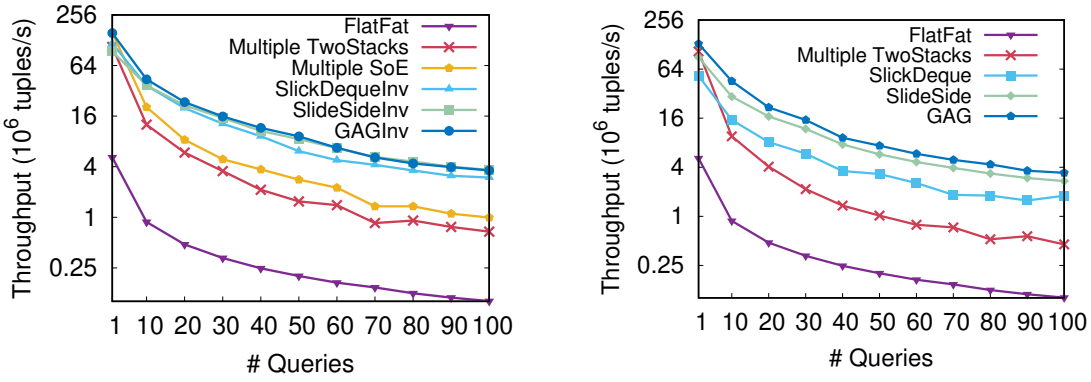


Fig. 4.21 Latency for 16K tuples window size



(a) Multi-query throughput (sum)

(b) Multi-query throughput (min)

Fig. 4.22 Comparison of incremental processing techniques for multiple queries

Figure 4.20a 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 function with the same window sizes as before (SM_{min}). Figure 4.20b 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 one tuple, 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). Figure 4.21 shows that our approach exhibits the lowest latency in min, max, median, 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.

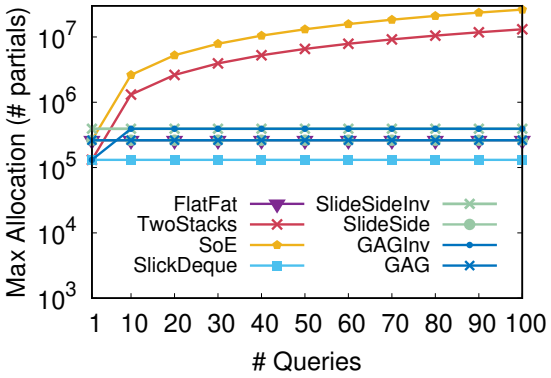


Fig. 4.23 Memory consumption

Multi-query evaluation. In these experiments, we generate multiple queries with uniformly random window sizes in the range of $[1, 128\text{K}]$ of tuples). The window slide for all queries is one tuple, which constitutes the worst case. We create workloads with 1 to 100 concurrent queries. For invertible functions, we use $\mathsf{SM}_{\mathsf{sum}}$ and, for non-invertible ones, we use $\mathsf{SM}_{\mathsf{min}}$. For TwoStacks and SoE, we replicate their data structures for each window definition because we cannot use them to evaluate multiple queries out-of-the-box.

For invertible functions shown in Figure 4.22b, GAG has comparable performance to SlideSide and outperforms SlickDeque by 45%. In Figure 4.22b, 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 Figure 4.23), 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.

Discussion. In summary, GAG captures the performance behavior of the state-of-the-art incremental algorithms and generates code that achieves the highest throughput and lowest latency in all evaluated scenarios. We, therefore, designed an abstraction that allows SPEs to exhibit robust performance irrespective of the window definition and the aggregation function type of the executed queries. Our approach demonstrates higher consumption regarding the memory requirements, only for multi-query workloads. In this case, GAGs trade-off performance with memory by requiring at most 3× more partials than the next best performing approach. However, based on the benchmark queries from above, the number of partials is in the order of hundreds.

4.8 Limitations and discussion

In this section, we highlight some of LIGHTSABER's current limitations and initiate a discussion on how they can be addressed as part of future work.

Optimizing scale-up execution. Our focus in LIGHTSABER is scaling window aggregation queries on multi-core architectures. Therefore, an interesting future direction is to utilize other types of parallel hardware, such as GPUs [137] or FPGAs [208], to accelerate streaming applications. In terms of parallel join implementations, we want to explore how existing solutions [237] integrate with our execution model. Furthermore, given that LIGHTSABER generates code that worker threads execute based on data locality, we want to study more advanced solutions for complex CPU topologies similar to recent work [236]. Finally, it would be interesting to investigate how we can extend our system design to handle out-of-order event processing [158] and applications with transactional [156] or iterative semantics.

Distributed and out-of-core execution. In this work, we addressed the challenge of accelerating latency-critical streaming applications with intermediate operator state that fits in single-node multi-core architectures. However, such an assumption may not be practical for several scenarios, resulting in a distributed processing model [24, 15, 80, 211] that exploits the data-parallelism on a shared-nothing cluster. Given that scale-out approaches could significantly impact the complexity of the system and the end-to-end processing latency, we want to explore a novel out-of-core design that offloads operator state to local or remote disks and non-volatile memory to handle such applications.

Window aggregation. For in-memory relational window operators, Leis et al. [144] propose a general algorithm that utilizes intra-partition parallelism for large hash groups and a specialized data structure for incremental computation. However, this work neither exploits the parallelism and incremental computation opportunities nor is expressive enough to support time-based windows as LIGHTSABER. Compared to distributed window aggregation, existing solutions perform redundant operations [33] or approximate window semantics [231] due to the lack of a global clock for synchronization. Distributed SPEs such as Storm [211], Spark Streaming [231] or SEEP [80] do not respect window semantics by default and require extensions [213] or custom solutions, which are error-prone. Millwheel [6] supports rich window semantics, but it assumes partitioned input streams and does not compute windows in parallel. Finally, a noteworthy future direction for LIGHTSABER is the extension of its model to incorporate other window types (i.e., session windows [7]).

Incremental execution. Recent work on window aggregation [23, 202, 200, 194, 38, 207] 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 [201], to future work. GAGs can handle FIFO windows with "in-order" or "slightly out-of-order" events that end up in the same partial aggregate. Panes [146], Pairs [140], Cutty [46], and Scotty [212] are different slicing techniques, which are complementary to our work—LIGHTSABER can generate code to support them.

Before concluding this section, let us discuss the conditions under which GAGs are applicable. In terms of the aggregation types, the aggregate functions must be only associative, similar to existing approaches (see Table 2.1). 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 [106]. GAGs can handle holistic functions by inserting a "trigger point" for every tuple insertion, which is, however, inefficient compared to other approaches [105, 104]. New functions can be added similar to recent work [202, 46] using the interface from Section 2.3.5. The collapsing of the GAG to SoE requires an extra `invert` function during the specialization phase.

4.9 Summary

To achieve efficient window aggregation on multi-core processors, SPEs need to be designed to exploit 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 and incremental processing opportunities.

We developed two novel abstractions to implement this design, 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 (*GAG*s)*, 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 LIGHTSABER. It outperforms state-of-the-art systems by at least a factor of two on our benchmarks. Some benchmarks even show improvement beyond an order of magnitude.

We consider LIGHTSABER a comprehensive solution towards a *general-purpose* relational SPE that exploits multi-core architectures. However, given the nature of single-node execution, we still need to address the challenge of fault-tolerance to make such a design

practical for real-world use cases. In Chapter 5, we introduce a novel system to achieve single-node scale-up fault-tolerance.

Chapter 5

Accelerating Fault-Tolerant Stream Processing on a Single Node

In the previous chapter, we described a tree abstraction (*parallel aggregation tree*) and a graph abstraction (*generalized aggregation graph*) that allow SPEs to balance parallelism and incremental processing for stream queries. We combined these abstractions to design LIGHTSABER, a state-of-the-art SPE that uses JIT query compilation and rivals the performance of cluster-based deployments. While LIGHTSABER addresses the high throughput and low latency performance requirements [186, 166] on multi-core architectures, our design faces limitations for reliable stream processing. In particular, LIGHTSABER, similar to existing scale-up solutions, does not provide deterministic results or data integrity guarantees upon failures [197, 15, 118]. However, producing consistent and repeatable results is a crucial stream processing requirement (see Section 2.1.1), especially when considering the expected failure rates in large data centers [91] nowadays.

This chapter describes the fault-tolerance aspects of a modern, reliable scale-up SPE and, more precisely, focuses on designing such systems with exactly-once semantics. However, providing reliable stream processing on a single-node deployment without sacrificing performance remains an open challenge. First, due to the limited I/O bandwidth of a single node, it becomes infeasible to persist all stream and operator data of continuous applications. Second, existing fault-tolerance mechanisms rely on distributed systems [16] to recover stream and operator state upon failures. These systems require cluster-based deployments, leading to a higher resource and maintenance footprint. With single-node SPEs lacking built-in fault-tolerance mechanisms, their adoption in real-world scenarios is hindered.

As discussed in Section 2.5, existing SPEs achieve at-least-once or exactly-once delivery semantics by persisting streams and processing state [45, 80, 230]. These persistence operations are offloaded to external distributed messaging systems [16, 11, 180] or key-value

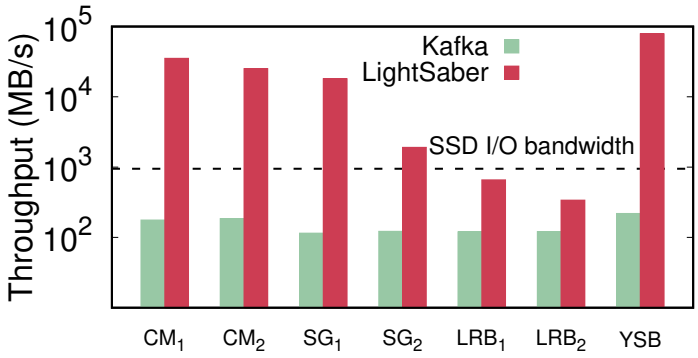


Fig. 5.1 Data ingestion rates for stream queries in single-node SPE (LIGHTSABER) vs. persistent message queue (Apache Kafka)

stores [78, 50, 53], which, however, induce performance overheads [185, 183, 72] and lead to scaled-out deployments.

To illustrate the magnitude of the problem, Figure 5.1 shows the difference in ingestion throughput for a set of real-world stream queries [205] (see Section A.1) between LIGHTSABER [205], a high-performance single-node SPE with query compilation, and Kafka, a popular persistent message queue system. As the results show, a single Kafka node can only ingest data streams at rates that are several orders of magnitude lower than LIGHTSABER's query performance and does not even saturate the SSD bandwidth (indicated by a dashed line). However, addressing the problem by scaling out the Kafka deployment counteracts the benefits of a single-node deployment.

At the same time, the strawman solution of persisting all input streams and processing state to stable storage is impractical. For a set of real-world stream queries [205] in Figure 5.1, disk I/O bandwidth becomes the limiting factor trailing performance to 950 MB/s. While hardware solutions (e.g., NVMe SSDs [226] or RAID [170]) can increase bandwidth at the expense of additional costs, in this chapter, we argue that *a modern SPE should provide reliable stream processing mechanisms without increasing the resource footprint or affecting processing performance*.

Opportunity. The emergence of scale-up SPEs [137, 158, 236, 205, 157] that compete with the performance of distributed systems provides an unprecedented opportunity for rethinking the design of reliable single-node stream processing mechanisms. To prevent expensive scale-out approaches while retaining predictable latency for time-critical applications, we need to revisit the following mismatches: (i) the limited single-node resources make existing fault-tolerance mechanisms impractical; (ii) messaging systems, such as Kafka, do not provide efficient storage and, thus, cannot saturate the bandwidth of modern SSDs (indicated

by a dashed line in Figure 5.1); and (iii) while high-speed networking [41, 125, 36] enable fast stream ingestion and remote storage [135], existing SPEs cannot saturate these fast interconnects [234].

Requirements. To design a *reliable* relational single-node SPE, our system must must address the following requirements (**R1**-**R3**):

1. *Reduce* the *required I/O bandwidth* irrespective of the workload. Given the limited available single-node disk bandwidth, an efficient SPE must make workload-aware decisions to accelerate persistence.

2. Ensure *high throughput* and *low-latency* results during execution without failures. To guarantee *high resource utilization*, an SPE must interleave persistence and network communication with query execution.

3. Provide *fast recovery* upon failure. Given the single-node nature of query execution, it is crucial to provide sub-second recovery latencies to lower the system's downtime.

In this chapter, we describe SCABBARD, the first single-node SPE that supports exactly-once fault-tolerance semantics despite limited local I/O bandwidth. First, we present an overview of SCABBARD and its components in Section 5.1. We, then, introduce our data and fault-tolerance models in Section 5.2. In Section 5.3, we discuss how SCABBARD integrates persistence operations with the query workload through a set of novel abstractions and optimizations to achieve efficient persistence (requirement **R1**). Subsequently, in Section 5.4, we describe how SCABBARD further reduces persisted data volume (**R1**) using workload-specific compression: it monitors stream statistics and dynamically generates computationally efficient compression operators. In Section 5.5, we discuss SCABBARD's design and implementation (**R2** and **R3**), followed by our experimental evaluation (Section 5.6), the limitations of our solution (Section 5.7), and summary (Section 5.8).

5.1 Overview

Our goal is to design and implement a single-node fault-tolerant SPE whose fault-tolerance mechanism (i) accounts for the limited available I/O bandwidth (**R1**), especially when using remote storage [10]; (ii) has a low impact on processing performance without failures (**R2**); and (iii) allows fast recovery after failures (**R3**). Our key idea is to reduce the required disk I/O bandwidth by tightly integrating *stream* and *state persistence* with the *operator dataflow graph* of the query. This way, the SPE can apply workload-specific optimizations to (a) reduce

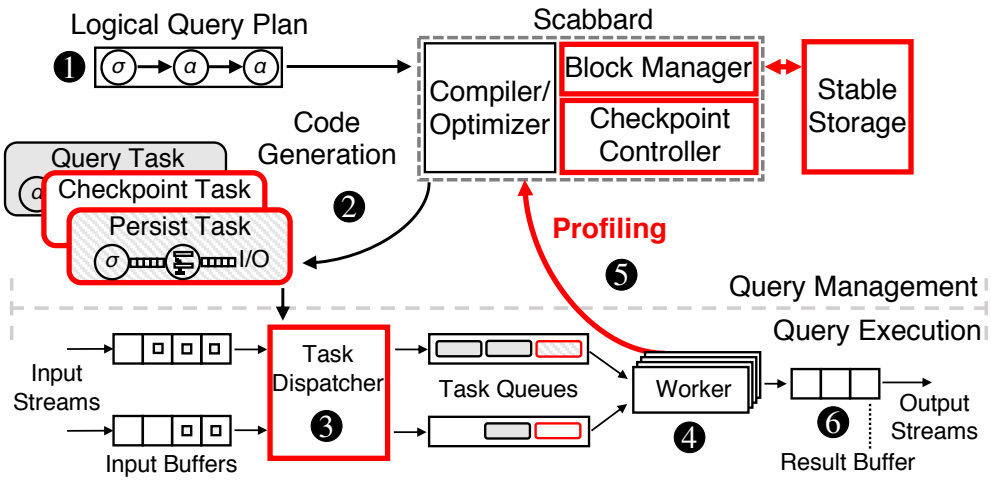


Fig. 5.2 SCABBARD architecture

I/O bandwidth by persisting stream and operator state only after high selectivity operators have executed and (b) compress data before persistence with query-specific compression.

To address requirements **R1**-**R3**, we introduce SCABBARD, a new single-node fault-tolerant SPE that provides exactly-once semantics without compromising processing throughput. SCABBARD's query execution engine is based on LIGHTSABER, which we describe in Chapter 4, and its fault-tolerance approach is to persist input streams and transient operator state to a local or remote SSD. SCABBARD introduces a novel *persistent operator graph* model and *adaptive compression* techniques that enable *workload-aware* persistence. At the same time, to realize efficient data storage and fast recovery, it introduces a *Block Manager* and a *Checkpoint Controller* that orchestrate persistence operations and recovery. Next, we discuss the key components of our design shown in Figure 5.2,[1] highlighting in red the features introduced by SCABBARD.

Imperative API. SCABBARD extends the imperative API of LIGHTSABER to enable users to express relational stream window queries [21] with exactly-once semantics upon failure.

[1]For simplicity, the figure omits the interaction of the query execution layer with the Block Manager and the Checkpoint Controller.

Example: In Figure 5.3, we illustrate an example of a fault-tolerant query pipeline with a SELECT operation and highlight the changes required to LIGHTSABER's applications (see Figure 4.3) to transform them to fault-tolerant. In particular, in line 7, we specify that the SELECT operator is considered fault-tolerant. Then, in line 12, we determine that the query's input stream requires persistence due to the absence of a fault-tolerant stream source. Finally, in line 13, we initialize this pipeline fragment [165] with a predefined compression scheme. We explain how SCABBARD performs workload-aware compression in Section 5.4.

```
1    auto predicate = new ComparisonPredicate(EQUAL_OP, new ↩
     ColumnReference(1), new IntConstant(3));
2    auto selection = new Selection(predicate);
3    auto window = new WindowDefinition(ROW_BASED, 60, 60);
4    auto cpuCode = new OperatorKernel()
5                            .setInputSchema(getSchema())
6                            .setSelection(selection)
7                            .setFaultTolerant()
8                            .setup();
9    auto persistInput = true;
10   auto query = new Query(new QueryOperator(*cpuCode),
11                           *window,
12                           persistInput);
13   query->setCompression([](...)-> {...});
14   auto application = new QueryApplication(query)
15                           .setup(NUM_OF_WORKERS);
16   while (true) {
17     input = ...;  /* get a pointer to a batch of data */
18     application->processData(input);
19   }
```

Fig. 5.3 Imperative API for SCABBARD (SELECTION operator example)

Compiler. SCABBARD extends LIGHTSABER's compiler to perform JIT query compilation based on the following steps: (i) it parses the execution graph defined by the imperative API in stage ❶; (ii) it translates the execution graph into a *persistent operator graph* (POG), which enables *workload-aware* decisions for data persistence, and performs optimizations using a set of logical rules [218, 21, 107] in ❷; and (iii) it code-generates *persistence*, *checkpoint*, and *query* tasks using the optimized plan in stage ❸. Let us now focus on the last two steps of the compiler by introducing our *persistent operator graph model* and then discussing the mechanism of code-generating *query-specific adaptive compression*.

In Section 5.3, we introduce a new *persistent operator graph* model that allows for *workload-aware* decisions about data persistence: operators can be reordered to reduce the required I/O bandwidth for persistence. Based on query characteristics, an SPE can "push per-

sistence up",[2] i.e., pruning data with high-selectivity operators. To enable parallelism when persisting streams and operator state, the persistent operator graph uses two compile-time abstractions: persistent streams, or *p-streams*, and fault-tolerant operators, or *ft-operators*. P-streams are reliable FIFO channels that support the parallel logging of streams; ft-operators enable the parallel checkpointing and recovery of stateful operators' state.

In Section 5.4, we examine how to reduce the persisted data further using p-streams adaptive compression. To achieve this, an SPE generates custom compression operators in ❺, taking stream statistics (e.g., data ranges, sequences of equal values, etc.) into account. This exposes the trade-off between the computational cost of a compression algorithm and the compression benefit in terms of saved I/O bandwidth. The SPE then selects a suitable compression algorithm (e.g., run-length encoding, null suppression, delta-encoding, etc.) and inserts compression operators dynamically into the persistent operator graph. The choice of compression algorithm is adaptive: when the statistics of the p-stream change, SCABBARD switches to a new compression algorithm while processing.

Runtime. In Section 5.5, we discuss how SCABBARD provides a runtime that manages query execution, consistent checkpointing, and recovery on multi-core CPUs during stages ❸, ❹, and ❻. It extends LIGHTSABER's runtime using control tuples (*markers*) from the *POG* to coordinate persistence decisions. These *markers* flow between operators and trigger storage and garbage collection operations. At the same time, SCABBARD introduces a *Block Manager* and a *Checkpoint Controller* that orchestrate persistence and recovery operations.

To achieve sub-second recovery latencies, SCABBARD reduces the data loaded from storage. It persists the ft-operator state frequently with low overhead through asynchronous checkpointing. Furthermore, it avoids the overhead of query compilation during recovery by storing the optimized code of compiled queries in a native binary format. To recover only the minimum data, persisted data is garbage collected when the dependent results have been emitted or persisted.

SCABBARD engine. Based on the above components, we design and implement SCABBARD, the first single-node SPE that supports exactly-once fault-tolerance semantics on multi-core CPUs. Our evaluation shows that SCABBARD introduces less than 30% overhead in processing throughput compared to execution without fault-tolerance. On a 16-core server, it processes over 200 million tuples per second with 8 ms latency (95^{th} percentile) and recovers below a second. It outperforms Apache Flink, a state-of-the-art fault-tolerant SPE, by at least an order of magnitude for all our benchmarks. SCABBARD achieves stream persistence similar to a 20-node Kafka cluster with 3× lower 95^{th} percentile latency. It achieves a throughput of up to 10.5 GB/s using 100 Gb/s InfiniBand with RDMA for stream ingress.

[2] We use a relational view in which "up" means closer to the output.

5.2 Stream processing and fault-tolerance models

Before presenting the compile-time and runtime abstractions of the *POG* model, it is necessary to revisit the definition of the data, operator, and failure models provided in Section 2.1.2 and Section 2.5.1. We start by extending both data and operator models to express the fault-tolerance mechanisms introduced next.

Data model. In this work, we also adopt a relational stream model with *windows* that follows the semantics of the *continuous query language* (CQL) [21].

Definition 5.2.1 (data stream). A *stream* s is an infinite sequence of *tuples*, $t \in s$. Each tuple $t = (\varepsilon, \tau, p)$ has: an event timestamp $\varepsilon(t) \in \mathscr{E}$ that denotes when the event occurred, where $\mathscr{E}$ is an ordered time domain of discrete non-negative integer values; a logical timestamp $\tau(t) \in N^+$ assigned by a monotonically increasing logical clock at each operator upon receipt [80]; and p, a sequence of values of primitive data types.

We assume that tuples in a stream adhere to a *temporal ordering*, i.e., arrive *in-order* based on their event timestamps.

Operator model. Each operator o receives tuples from n upstream operators to its input queues, $I = \{s_1, ..., s_n\}$. It then applies a deterministic operator function f, and produces tuples for its downstream operators stored in a result buffer, denoted by R. An operator keeps track of exchanged tuples with two progress vectors, PV^{in} and PV^{out} [222], while stateful operators have processing state Θ. For ease of presentation, we denote an operator snapshot as $C = (I, R, \Theta, PV^{in}, PV^{out})$ and use the notation C_{τ_e} to indicate that it has all values up to τ_e.

Every operator function f is composed of: (i) a state transition function ρ that accepts the current state Θ_i and an input tuple t_i and yields the new state Θ_{i+1}; and (ii) an output function ω that accepts a state and an input tuple and outputs one or more[3] tuples $\langle t_j, ..., t_{j+x} \rangle$.

We consider queries that use *window* functions over streams to transform them into finite sequences, called *window fragments* [137]. For efficient operator parallelization [205, 137] without depending on distinct keys, every state transition function is decomposed into: (i) a *fragment* function ρ^f that processes a sequence of fragments and produces immutable partial results; and (ii) an *assembly* function ρ^α that constructs and reorders complete window results. Each operator generates computational tasks by bundling fixed-sized data *batches* from its inputs with ρ^f and ρ^α functions [4].

Failure model. Regarding the failure model, we adopt the assumptions discussed in Section 2.5.1: (i) non-deterministic software or hardware failures [134] cause an SPE node

[3] e.g., join operators produce multiple output tuples per input tuple

[4] See previous work [137] for details of how different operators are decomposed into ρ^f and ρ^α.

```
(select timestamp, vehicle, highway, direction, segment, count(*)
from SegSpeedStr [range 30 slide 1]
group by highway, direction, segment, vehicle) as R
--
select timestamp, highway, direction, segment, count(vehicle)
from R
group by highway, direction, segment
```

Fig. 5.4 LRB_3 query in CQL

to *fail-stop* [75]; (ii) the SPE is connected with external sources and sinks via high-speed networking that provides a reliable FIFO delivery protocol; and (iii) the SPE has access to stable storage that survives failures. We assume deterministic operators and "in-order" input streams as mentioned above. Finally, Definition 2.5.1 provides the operator graph representation of our model.

5.3 Scale-up fault-tolerance using persistent operator graphs

Having presented our fault-tolerance assumptions, we now introduce the persistence mechanisms required to perform the storage and recovery operations in a single-node SPE. As discussed in Section 2.5, to resume processing from the point of failure and achieve *exactly-once* results, an SPE must track the query's execution progress and recover the state of *stateful* operators. This process demands the redundant storage of: (i) the input streams to rebuild the lost operator state through data replay; (ii) the computational state [45, 80] or logic [230] to avoid replaying all state dependencies (e.g., entire stream history for global windows or hours of data for large time-based windows).

While performing these redundant storage operations efficiently in a cluster-based solution has been studied extensively [211, 6, 80, 24], a single-node SPE faces several limitations in terms of disk bandwidth, disk space, and CPU capacity. Therefore, its fault-tolerance mechanisms must persist only the required parts of streams and operator state to enable recovery. However, selecting what to store or how to manage persistence and recovery operations are non-trivial tasks, which are highly query- and input-specific. In particular, which data to persist and discard cannot be determined statically and instead requires runtime knowledge about e.g., the data lifetime and its distribution.

Our idea is to exploit the information of a stream query to enable optimizations related to persistence by encoding it in a structure that we call a *persistent operator graph* (*POG*). The POG contains aspects of both query compile-time and runtime. Figure 5.5 shows the

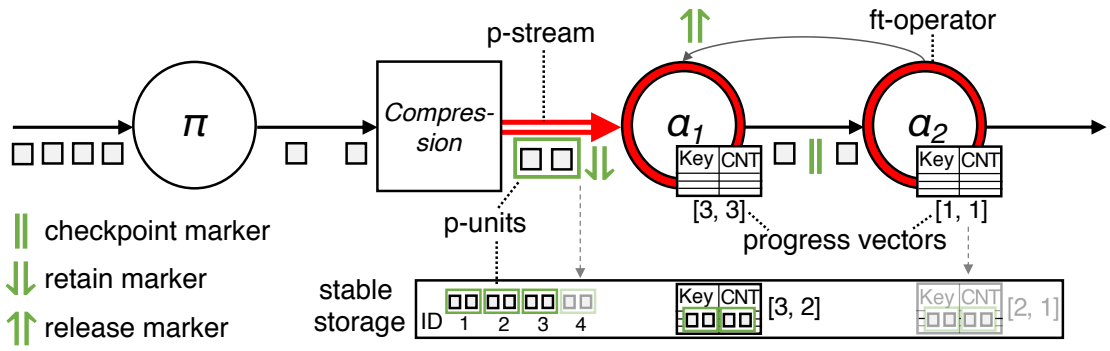


Fig. 5.5 Persistence operator graph (POG) with p-streams, ft-operators, and markers for LRB_3 query

POG for the third Linear Road Benchmark query [22] (defined in Section 5.6.1, listed in Figure 5.4). The POG extends the operator graph defined in Sections 2.1.2 and 2.5.1 with two new compile-time abstractions (shown in red) and two coordination abstractions (shown in green). Through its compile-time abstractions, *persistent streams* (*p-streams*) and *fault-tolerant operators* (*ft-operators*), a POG supports stream and state persistence. As these persistence operations require runtime coordination to achieve consistency when recovering, the *POG* also introduces *persistence units* (*p-units*) and a set of coordination *markers*. A p-unit is a batch of data (i.e., a finite subsequence of a stream or operator state, usually around 512 KB) with associated metadata (i.e., lineage information). Persistence, recovery, and garbage collection operations on each p-unit are performed atomically. Markers are special control tuples in a stream [44, 52], similar to *watermarks* discussed in Section 2.1.2, that coordinate the flow between operators and trigger persistence and removal operations.

5.3.1 Compile-time abstractions for persistence

The compile-time abstractions of POGs expose operations for the persistence management of streams and computational state in an operator graph, which can be executed in parallel at runtime. These abstractions are accessible through intuitive interfaces that we summarize in Table 5.1 and describe next:

A **persistent stream (p-stream)** provides a reliable FIFO communication channel between two operators in the POG, supporting asynchronous stream persistence at the granularity of p-units. For example, p-streams can be used as ingress streams to persist the incoming data in the absence of a reliable stream source. Every p-unit in a p-stream is assigned a monotonically increasing logical timestamp $\tau(t)$ upon insertion, which maps to the offset from the logical position of the first tuple in the stream. If a p-stream is marked for

Table 5.1 SCABBARD fault-tolerance abstractions

Entity	Definition in C++ notation	Description
PStream	`void subscribe(Operator, ReaderId, Offset)`	Subscribes an operator as reader to the channel
	`promise<void> write(PUnit<Tuple>, BatchId, Offset, isPersistent)`	Adds data to the channel
	`PUnit<Tuple> read(ReaderId)`	Returns the next available tuples
	`promise<void> checkpoint(BatchId, CheckpointId)`	Writes data to stable storage
	`void trimFrom(ReaderId, Offset, isPersistent)`	Removes data from the channel
	`promise<void> loadStream(PUnit<Tuple>, BatchId)`	Loads data to the channel from stable storage
FTOperator	`void prepareCheckpoint(CheckpointId)`	Mark the streams/state that are going to be checkpointed
	`promise<void> checkpoint(PUnit<Tuple>, BatchId, CheckpointId)`	Writes state to stable storage
	`promise<void> loadState(PUnit<Tuple>, BatchId, CheckpointId)`	Loads state from the last valid checkpoint
	`void sendAck(FTOperator)`	Sends a retain marker to an upstream operator
	`void setDependencies(PUnit<Tuple>)`	Calculates the data dependencies of a PUnit
	`list<Offset> getLatestOffsets()`	Returns a list of the latest offsets it has received
PUnit	`void compress((...)->{...}, StorageBuffer, Index)`	Compresses tuples with a given compression function
	`void decompress((...)->{...}, StorageBuffer, Index)`	Decompresses tuples with a given decompression function
	`list<Offset> getDependencies()`	Returns a list of data dependencies
POG	`void insertRetainMarker(PStream)`	Sends a retain marker to a channel
	`void insertCheckpointMarker(FTOperator)`	Sends a checkpoint marker to an operator

persistence with coordination markers (see Section 5.3.2), its data becomes available only after disk storage to provide consistent results.

While a p-stream is related conceptually to the well-known notion of *upstream backup* [112], upstream backup persists only ingress streams by utilizing an external system to create backups. In contrast, a POG allows persistence anywhere in the operator graph, which enables new optimizations later introduced in Section 5.3.4.

The p-stream interface allows to `subscribe` to, `write` to and `read` from its channel as shown in Table 5.1. As multiple operators can subscribe to a single p-stream, a p-stream must track their progress using a stream `Offset` for every subscriber.

A **fault-tolerant operator (ft-operator)** extends streaming operators with support for consistent checkpointing and recovery through *progress tracking*. In Table 5.1, we describe its interface to create `checkpoints` and `loadState` from the last consistent snapshot. In general, ft-operators can be stateless ($\Theta = \emptyset$), e.g., `PROJECTION` (π), `SELECTION` (σ), or stateful with `WINDOW` semantics, e.g., `AGGREGATION` (α), `GROUP-BY` (γ), `JOIN` ($\bowtie$). For stateful operators, the ft-operator partitions its state Θ into *immutable* p-units, which can be persisted to and recovered from storage atomically.

However, tracking data dependencies poses a challenge, as a p-unit may contribute to multiple results (e.g., in the case of sliding window aggregation). An ft-operator solves this problem by computing the dependencies between p-units: it attaches a lightweight graph structure using `setDependencies`. The p-unit dependencies are calculated based on the logical timestamps, the input ordering, and the window semantics (similar to Timestream [181] or D-Streams [231]). Therefore, these dependencies capture the relationship: (i) between p-units from different streams (i.e., stream-to-stream dependencies); and (ii) p-units from state and streams (i.e., state dependencies). The logical timestamps of the graphs can be serialized to vector clocks *VC* [153], which determine the event ordering upon recovery.

Although every deterministic operator in the POG can become an ft-operator, checkpointing overhead can be traded off against recovery time by replacing only the most downstream operators with ft-operators. Depending on the window semantics (e.g., sliding windows) or operator selectivity (e.g., `JOINs`), it may be preferable to avoid checkpointing. To prevent inconsistent operator state after a failure [75], if an operator is marked as fault-tolerant, the POG replaces all its downstream operators with ft-operators. Without this strategy, checkpoints do not ensure recovery to a consistent state and require additional logic upon failures or during garbage collection (see Section 5.3.3). Next, we formally define POGs, each consisting of data streams and operators that can be fault-tolerant, and describe in detail the POG of Figure 5.5.

Definition 5.3.1 (Persistent operator graph). A POG is represented as an operator graph similar to Section 2.5.1, i.e., $q = (O, S, B)$. While a node $o \in O$ can be an ft-operator only if all its downstream operators are fault-tolerant, all POG's edges $s \in S$ can be p-streams.

Example: Figure 5.5 shows an instance of POG with four operators: the stateless `PROJECTION` and `COMPRESSION` and two stateful grouped `AGGREGATIONS`. Only the stateful operators (i.e., a_1 and a_2) are marked as fault-tolerant, while the communication channel between `COMPRESSION` and a_1 operator is a p-stream. By *pushing-up* persistence further from the input stream, POGs manage to reduce disk bandwidth (we discuss how they enable such optimizations in Section 5.3.4). Regarding POG's runtime components, Figure 5.5 shows how both the data stream and operators' state are partitioned in p-units and stored to disk using monotonically increasing identifiers. The following section discusses how POGs utilize the different types of markers (depicted in green) to coordinate fault-tolerant operations.

5.3.2 Persistence and recovery coordination

After a failure, the SPE must recover and recompute the data required to recreate the POG's operator state. However, it is challenging to provide exactly-once semantics when recovering, as it involves runtime knowledge about the query's progress. To manage the operations required to achieve this on a single-node SPE, we introduce a *persistence protocol* using *markers*. *POGs* support three types of markers: (i) *checkpoint* markers trigger operator checkpoints; (ii) *retain* markers mark a p-unit in a p-stream for persistence; and (iii) *release* markers signal that a specific p-unit is no longer required for recovery. The *persistence protocol* uses consistent snapshots to perform state recovery. Following the state recovery, all data that is not part of the last checkpoint must be replayed while tuples already produced are dropped. The persistence protocol has five asynchronous primitives, shown in Algorithms 8 and 9 that are described next.

Consistent checkpoint coordination is achieved by *checkpoint* markers similar to the Chandy-Lamport algorithm [52]. These markers are injected with `insertCheckpoint-Marker` at regular intervals or using a custom dynamic trigger. When an ft-operator receives a checkpoint marker (line 7 of Algorithm 8): (i) on the first invocation, the `prepareCheckpoint` function triggers the synchronous *prepare phase* for all the p-units of the operator's transient state Θ and its queues I and R; and (ii) once all checkpoint markers are received from its upstream operators, the operator creates and dispatches asynchronous tasks to write the p-units to storage. We decide to accelerate persistence with asynchronous I/O operations because they can overlap with CPU operations (i.e., query execution). The checkpoint completes when all marked p-units have been persisted.

Algorithm 8: POG's persistence protocol executed by operator o

```
1  init                                                          ▷ Initialize local variables
2      C = (I, R, Θ, PV^in, PV^out) ← ({∅}, ∅, ∅, {0}, {0})
3      U ← {o_i, ..., o_{i+x}}, D ← {o_j, ..., o_{j+y}}          ▷ Upstream and downstream operators
4      snapshot ← ∅, marked ← ∅, taskQueue ← ∅, persist ← {false}
5  upon receive < marker > from in ∈ I
6      (type, VC) ← marker
7      if type = checkpoint then
8          marked ← marked ∪ in
9          if |marked| = 1 then                                  ▷ The first marker overtakes all t ∈ I or R
10             broadcast(D, marker)
11             snapshot ← snapshot ∪ I ∪ R ∪ Θ
12         if marked = I then                                    ▷ Store to disk when all markers are received
13             taskQueue ← taskQueue ∪ checkpointTasks(snapshot)
14             snapshot ← ∅, marked ← ∅
15     else if type = release then                               ▷ Remove obsolete data from C
16         C = C \ C_{VC[D]}, broadcast(U, marker)
17     else persist[in] ← true
18 upon receive < punit > from in ∈ I
19     (tuples, offset, VC) ← punit
20     if offset > PV^in[in] then                                ▷ Persist channels that have not sent a marker
21         if |marked| ≠ 0 ∧ in ∉ marked ∧ persist[in] = false then
22             snapshot ← snapshot ∪ punit
23         (Q, id, offset) ← in
24         taskQueue ← taskQueue ∪ persistTask(Q, tuples, id, offset, persist[in])
25         in ← (Q, id + 1, offset + |tuples|)
26         PV^in[in] ← PV^in[in] + |tuples|, persist[in] ← false
27 upon receive < notification > from in ∈ I
28     taskQueue ← taskQueue ∪ queryTask(ρ^f, ρ^α, I, R, Θ)
29 upon receive < notification > from R
30     for out ∈ D do                                            ▷ Simplified version of sending data downstream
31         (punit, offset, VC) ← read(R, out)
32         if offset > PV^out[out] then
33             ack ← send(out, punit)
34             if ack then
35                 PV^out[out] ← PV^out[out] + |tuples|
36                 if o = mostDownstream then
37                     marker ← (release, VC), broadcast(U, marker)
```

Algorithm 9: POG's recovery protocol executed by operator o

```
1 upon recovery
2     VC = loadMetadata() ⊕ requestMetadata(D)
3     broadcast(U, VC)
4     PV^in ← VC[U]
5     PV^out ← VC[D]
6     taskQueue ← taskQueue ∪ recoveryTasks(C)
```

Definition 5.3.2 (**At-least-once property**). Given a global checkpoint GC_{τ_e} of graph q at τ_e and a snapshot $C^{o_i}_{\tau_{e_i}}$ of operator o_i at τ_{e_i}, the Chandy-Lamport algorithm guarantees that every tuple from GC_{τ_e} is captured either in the upstream operator's queue or the downstream operator's queue or state: $\forall C^{o_1}_{\tau_{e_1}} \in GC_{\tau_e} \forall C^{o_2}_{\tau_{e_2}} \in GC_{\tau_e}, (o_1, o_2) \in S \forall \tau_n : (\tau_n \leq \tau_{e_1} \Rightarrow t^{o_1}_n \in R^{o_1}) \wedge (\tau_n > \tau_{e_1} \Rightarrow t^{o_1}_n \in I^{o_2} \vee \tau_{e_2} \geq \tau_n)$. We refer to that as the *at-least-once property*.

Efficient data replay. The persistence protocol replays tuples that are not captured in the last consistent checkpoint by determining what data to persist and remove with *retain* and *release* markers, respectively. The *retain* markers are injected into the ingress p-streams at periodic intervals using the POG's `insertRetainMarker` function. They flow through the POG (e.g., Figure 5.5 shows a retain marker between operators `COMPRESSION` and a_1). Upon receipt of a retain marker (line 17, Algorithm 8), the p-stream creates a sequence of p-units with the tuples that follow. To remove parts of the p-stream no longer required for the output result, the POG provides *release* markers (line 15), which are sent on feedback channels to discard p-units. This is shown in Figure 5.5, where a release marker is being sent from operator a_2 to a_1.

Data deduplication is achieved by exploiting the data dependencies between p-units to track the query progress and remove duplicates (line 32). As operators are deterministic and data are assigned monotonically increasing logical timestamps, filtering data already captured in the operators' progress vectors becomes trivial. However, persisting the dependency graphs for all operators before execution introduces a substantial overhead.

Therefore, to achieve exactly-once output, the persistence protocol persists only the most downstream operators' progress with two different methods. The first method requires a transactional sink [139, 180]. At the same time, the most downstream operator performs a *two-phase commit* to persist progress, which guarantees atomicity. The second approach requires the sink to store a serialized vector clock VC with every output result before making data available. The most recent vector clock is returned upon request and is used to filter duplicate tuples similar to line 20 of Algorithm 8.

To perform deduplication for single-input operators (e.g., `AGGREGATION`), assigning increasing logical timestamps to the p-units and using them to track progress is straightforward. For operators that ingest multiple streams, such as `JOINs` [126], before assigning logical timestamps, their inputs require to be buffered, ordered, and emitted deterministically based on the window semantics. An alternative approach for `JOINs` is to join the input streams greedily (i.e., non-deterministically without sorting), which accelerates execution but requires logging and replaying all join decisions upon failure. These decisions concern the order in which the p-units of both streams are joined.

The **recovery protocol** retrieves the query progress from storage before deciding which stream and state parts to restore. Recovery is divided into four phases (lines 1–6 of Algorithm 9): *progress recovery*, *upstream requesting*, *data recovery* and *upstream replay*.

All operators in progress recovery (line 2) load the timestamp intervals captured by their latest checkpoints and most recent committed dependencies. In upstream requesting (lines 2-3), operators send requests downstream for the latest persisted vector clock. At the end of this phase, every operator has sufficient information to reload the data from the last consistent checkpoint or streams and drop results already processed with its progress vectors (i.e., PV^{in} and PV^{out}). Next, the protocol moves on to the data recovery phase (line 6) using the `loadState` and `loadStream` functions, while parallelizing the process for effective hardware utilization. In the final upstream replay phase (line 27), operators send data downstream as they would during normal operation. This last phase transitions into regular execution as ingress streams receive new data.

5.3.3 Discarding obsolete data with garbage collection

In contrast to relational data processing, stream queries perform computation over infinite streams. The stream processing paradigm raises two challenges when persisting data: (i) the finite disk capacity, especially when considering faster non-volatile memory or the storage cost in a cloud infrastructure [136]; and (ii) the high recovery latency when replaying large amounts of data to ensure exactly-once semantics. Therefore, it is necessary to remove persisted tuples, state, and recovery metadata that is no longer required.

To discard obsolete data, an SPE can use garbage collection (GC) based on either retention policies [139] or classic mark & sweep. Neither approach, however, applies to a single-node SPE deployment: retention policies require the user to define a threshold for data removal for each query, which cannot be automatically derived from the query semantics; mark & sweep has a high runtime overhead [163]. Therefore, we propose a garbage collection approach that is *semantically partitioned* and *optimistic* under failure.

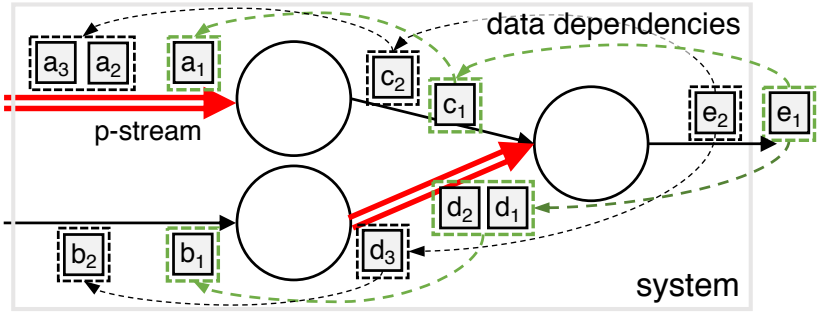


Fig. 5.6 P-unit dependency tracking

With *semantic partitioning*, a single p-stream or ft-operator retains ownership of each p-unit. Given that each operator tracks the dependencies of p-units and manages their ownership when passing them downstream, it is easier to reason about correctness when discarding data. *Semantic partitioning* also simplifies garbage collection under concurrency.

Given that p-units in streams are ordered, and the checkpoints capture the progress of ordered data and deterministic operations, garbage collection can be performed at a coarse granularity. This approach minimizes overhead because previous p-units with an `Offset` less or equal to a given value can be discarded in bulk. Without data loss, a p-unit can be removed if all its dependent results [141] have been either (i) persisted to disk or (ii) committed to the outside world. When these conditions are met, a reverse topological ordered traversal of the dependency graph is performed to send *release* markers upstream (lines 37 and 16). We refer to this garbage collection approach as *optimistic* because it guarantees that all p-units are eventually removed: when p-units with a larger `Offset` are discarded, they invalidate all previous p-units.

Example: Figure 5.6 shows the emission of tuple e_1 at which point its dependencies (shown in green) can be garbage collected. Note that all transitive dependencies of e_1 appear earlier than the dependencies of e_2 because p-units are ordered by monotonically increasing `Offsets` in their operator's partition. By using the `trimFrom` function, each operator removes obsolete stream and operator state data.

5.3.4 Optimizing POG with persistence push-up

We now describe the optimizations enabled by the POG to reduce disk I/O bandwidth and shorten the recovery process. Stream queries often consist of highly-reductive, inexpensive operators early on in their operator graphs, e.g., `SELECTION` or `PROJECTION`. These operators eliminate input tuples and significantly reduce the data volume that reaches stateful operators. Considering persistence as an operator node of the POG allows it to be "pushed up", i.e., exe-

cuted after the data reducing operators. This optimization provides a compact representation of the stream required for recovery and accelerates persistence.

To perform *persistence push-up*, the POG is traversed in topological order, and a set of transformation rules are applied to rewrite it. These transformation rules identify unused attributes and insert appropriate `PROJECTION` operators to prune them or push down selective operators (e.g., `SELECTION`). Persistence push-up is restricted to the stateless operator types that decrease the data size. Examples of such operators are: (i) operators with selectivity below one (e.g., `SELECTION` or a `HAVING` clause), i.e., ones that output fewer tuples than they consume; (ii) operators that reduce the number of bytes required to represent a tuple (e.g., `PROJECTION` or `COMPRESSION`).

Figure 5.5 shows a POG instance after persistence push-up, where the `PROJECTION` and `COMPRESSION` operators are placed to the left of the p-stream, thus decreasing I/O bandwidth for persistence. In particular, for query LRB_3, persistence push-up leads to $8\times$ disk I/O bandwidth reduction.

The rationale behind the choice of the previous operator types for *persistence push-up* is straightforward: stateful operators (e.g., `AGGREGATION` or `JOIN`) would amplify the output stream size based on the window semantics (e.g., for sliding windows with small slides), increasing the amount of stored data. Therefore, persistence push-up avoids pushing down stateful operators, as this would increase recovery latency and burden the external sources with buffering data for longer periods before the protocol acknowledges their persistence.

5.3.5 Persistence protocol correctness

Next, we formally show the correctness of the persistence protocol. The protocol considers the operator graph q as a single fail-stop recovery unit [198], i.e., if one or more operators fail, the whole graph must recover. The SPE has access to persistent storage that survives failures (i.e., remote block storage), allowing recovery to a different node. It communicates over reliable FIFO network channels with external sources/sinks to guarantee data delivery; the ingress channels[5] allow replay even under failure. To ensure exactly-once output, the protocol requires: (i) deterministic operators without side effects; (ii) a consistent checkpoint mechanism; (iii) and that the last externally committed tuple's vector clock VC is persisted.

First, let us prove that the persistence protocol guarantees exactly-once output for a single operator and then generalize this to arbitrary operator graphs. We start by defining the infinite sequence of all result tuples F based on the operator function definition from Section 5.2.

[5]For non-fault-tolerant external sources, the channels are replaced with p-streams.

Definition 5.3.3 (**Infinite tuple sequence F**). Each operator function is modeled as a pair of a state transition function ρ and an output function ω. F denotes the infinite (deterministic and correct) sequence of all tuples produced by an operator without failures. To restrict a tuple sequence to an interval, we use the notation $F[m,n]$ to denote $\langle \omega(\Theta_i, t_i) | i \in [m,n] \rangle$. We denote the deduplication function that uses logical timestamps to filter tuples as ϕ.

Theorem 1. *Given a single operator graph, failure at timestamp τ_f and recovery from timestamp τ_r, the persistence protocol produces a recovery sequence F_r, $F_r = \phi\left(F\left[0,\tau_f\right] + \left\langle \omega\left(\Theta_i,t_i\right) | i \in N, i \geq \tau_r \right\rangle\right)$ that is equal to the correct sequence, i.e., $F_r = F$.*

Proof. Let $C_{\tau_e} = (I, R, \Theta_{\tau_e}, PV^{in}, PV^{out})$ be the checkpoint at timestamp τ_e; let τ_p be the timestamp of the last persisted input tuple in the operator's p-streams such that $\tau_e \leq \tau_p$; let $X[\tau_{p+1}, \infty] = \left\langle t_i | i \in N, i \geq \tau_{p+1} \right]\rangle$ be the sequence of tuples held by the external sources (i.e., all tuples after τ_p); and let $VC = \langle \tau_{v_I}, \tau_{v_R} \rangle$ be the last committed vector clock at recovery time (τ_{v_I} and τ_{v_R} being the timestamps of input and output streams, respectively).

The first part of the recovery sequence $F[0,\tau_f]$ denotes the tuples emitted before failure, while $\langle \omega\left(\Theta_i,t_i\right) | i \in N, i \geq \tau_r \rangle$ denotes the sequence produced after failure. For the latter sequence, the operator retrieves its input sequences $I[\tau_{v_I}, \infty]$ and state Θ_{τ_f} in one of three ways: (i) if $\tau_p = 0$ (i.e., no data has been persisted and $\Theta_{\tau_f} = \emptyset$), all data is received from $X[\tau_{p+1}, \infty]$ and the recovery sequence becomes $F[\tau p + 1, \infty] = \left\langle \omega\left(\Theta_i,t_i\right) | i \in N, i \geq \tau_{p+1} \right]\rangle$; (ii) if $\tau_{v_R} \leq \tau_e$ (i.e., all output dependent to the checkpoint has been emitted), the operator reconstructs its state Θ_{τ_f} from an empty set by using the state transition function and replays all data persisted in its p-streams until τ_p. The remaining data is received from the sequence $X[\tau_{p+1}, \infty]$ and the operator uses the reconstructed Θ_{τ_f} to produce the sequence $\left\langle \omega\left(\Theta_i,t_i\right) | i \in [\tau_{v_I}, \tau_p] \right]\rangle + F[\tau p + 1, \infty]$; (iii) if $\tau_{v_R} > \tau_e$ (i.e., there is output that depends on the checkpoint), Θ_{τ_f} is restored from C_{τ_e} and data replay from the p-streams and used to produce the recovery sequence as in case (ii). Thus, in all three cases, F_r can be reconstituted from the last checkpoint and a finite external source buffer (i.e., only data between τ_p and the timestamp at the beginning of recovery).

Given that there may be overlap between the output before and after failure, the duplicate elimination function ϕ ensures at-most-once output. As the concatenated output stream before and after failure guarantees at-least-once, F_r equals F and has the exactly-once property. □

After proving the exactly-once output property for a single operator, we generalize it to arbitrary graphs.

Theorem 2. *Given an arbitrary execution graph with a single most-downstream operator o_d that is fault-tolerant,[6] a global coherent checkpoint GC, a failure at timestamp τ_f and*

[6]Decomposing a query with multiple outputs into multiple queries with a single output is straightforward.

recovery from timestamp τ_r, the exactly-once guarantee of the final operator extends to the entire operator graph.

Proof. Let us prove the theorem by induction, using Theorem 1 as the base case. The exactly-once fault-tolerance property of the graph is equivalent to the fault-tolerance of the most downstream operator o_d. Analogous to the single operator case, the fault-tolerance of o_d is proven for three cases: just as for Theorem 1, in cases (i) and (ii), the operator replays data from its inputs. By induction, the sequence produced by each input has the exactly-once property, even under failure; in case (iii), the operator loads its state from the last snapshot C_{τ_e} before triggering a downstream replay.

While the at-most-once guarantee stems from the ϕ-function, we must prove at-least-once processing of every input tuple, i.e., that every input tuple is either in its producer's output queue, the operator's input queue, or already reflected in the operator state. Formally, $\forall o_i \forall t_j \in F^{o_i} \exists \tau_x \,|\, \tau_x < \tau_f : \tau_j > \tau_x \Rightarrow t_j \in R^{o_i} \vee t_j \in I^{o_d} \vee \tau_e \geq \tau_x$. This follows trivially from the at-least-once property of a snapshot, as defined in Section 5.3.2. As o_d guarantees at-most and at-least once results, the operator graph guarantees exactly-once. □

Having discussed how stream execution graphs are transformed into POGs to allow efficient single-node fault-tolerance, let us now explain how SPEs can further reduce the required disk I/O through workload-aware compression in the following section.

Table 5.2 Compression algorithms

Name	**Description**
Base-delta [172]	Represents values as differences (deltas) from a base value
Delta-of-delta [173]	Delta-encoding over the delta-encoded data
Null suppression (NS) [189, 1, 13, 145]	Omits leading zeros from the bit representation
Simple-8b [13]	Stores integers in fixed-size blocks, first bits denote minimum values' bit-length
Variable byte (Var-Byte) [66]	Represents integers as variable number of bytes, using 1 status and 7 data bits per byte
Run-length encoding (RLE) [189]	Represents repeated sequences as pairs of values & counts
XOR compression [173]	Uses XOR'ed floating-point values
Dictionary [189, 1]	Data-agnostic compression scheme that replaces each value with a unique key from a dictionary
Snappy [241]	Dictionary encoding according to LZ77 [241]

5.4 Workload-aware stream compression

Since stream queries are long-running, it is beneficial to react to changing workload characteristics at runtime [31]. Therefore, we decided to reduce the required I/O bandwidth for

p-stream persistence using *adaptive compression*. Our approach considers the dynamic workload characteristics of data by monitoring p-streams and generating suitable compression operators. However, the best choice of a compression algorithm exposes a trade-off between compression ratio and throughput and depends on stream and query characteristics [67]. In this section, we first present a set of compression schemes (Section 5.4.1) and their performance characteristics by focusing on how to apply them to different data types (Section 5.4.2). We then analyze the process of adaptive compression and give examples on how to choose the most suitable algorithm (Section 5.4.3).

5.4.1 Lightweight compression for streaming data

Prior work in stream processing [171] shows that heavyweight schemes [221, 110] with high compression ratios are prohibitively expensive for real-time applications. Therefore, we consider lightweight techniques [1] that combine high-performance with resource efficiency, which we summarize in Table 5.2. Each lightweight compression algorithm takes as input a finite sequence of uncompressed values and produces a compressed representation (i.e., using as few bits as possible). The query processing model, however, affects the applicability of these algorithms: some approaches require either row or columnar format [43], and others can be applied to both. We next describe the compression schemes considered in this work:

- *Base-delta* encoding [172] represents each value as the difference (or delta) between that value and a reference value. It can be used with row and columnar format as input.
- *Delta-of-delta* encoding [173] applies delta-encoding over the delta-encoded data. Similar to *base-delta*, it is not affected by the query processing model.
- *Null suppression (NS)* [189, 1, 13, 145] omits leading zeros from values' bit representation. It can be applied to both input formats.
- *Simple-8b* [13] stores a series of integers in fixed-size blocks, with the first bits of each block denoting the minimum bit-length for its values. It is a specialized instance of *NS* that requires a columnar format.
- *Variable byte encoding (Var-Byte)* [66] represents an integer in a variable number of bytes, using 1 status and 7 data bits per byte. It is a specialized instance of *NS* that requires a columnar format.
- *Run-length encoding (RLE)* [189], which can be combined with other compression techniques, represents repeated sequences as a pair of the value and the number of repetitions (or runs). It requires a columnar format.

- *XOR* compression [173], uses XOR'ed floating-point values. It can be applied to both input formats.
- *Dictionary* encoding [1, 189] is a data-agnostic compression scheme that replaces each value with a unique key from a dictionary. It supports both input formats.
- *Snappy* [241] performs dictionary encoding according to LZ77 [241]. It can be applied to both input formats.

Most of the compression schemes presented above require a columnar format to be applicable or to exhibit better performance. In addition, the columnar representation leads to higher compression ratios. Therefore, we decided to store and compress data in columns that are more compressible [1, 3] but increase the decompression overhead. This decision results in faster execution when no failures occur, which we consider the typical case compared to the recovery process.

5.4.2 Exploiting workload characteristics

We base the decision which compression algorithm to use at runtime on three factors: (i) stream data distribution; (ii) compression ratio; and (iii) compression throughput. Figure 5.7 shows the associated performance trade-offs by plotting the compression ratio and throughput for different compression algorithms. Each line represents an input data type, and the marker location indicates the performance (in terms of throughput) for different algorithms. For example, the red line refers to a stream of timestamps; the markers show the compression ratio and throughput for each algorithm applicable to timestamp data. We next categorize the compression algorithms introduced above based on their applicability to different data types.

Timestamps. While timestamps are not a workload-specific data type, we consider them separately because they have discrete non-negative integer values with a relative order. In Figure 5.7, we explore RLE, Delta-of-delta, and Base-delta algorithms for two different distributions. If timestamps occur in fixed intervals (Timestamp 1), Delta-of-delta exhibits the best compression ratio and, thus, is used as the default. If multiple events occur within the same interval (Timestamp 2), Base-delta & RLE offers better performance.

Integers. We apply three compression schemes to integer types: Var-Byte, RLE with word-aligned NS (NS & RLE), and Simple-8b. With random not repeated values (Integer 1), Simple-8b achieves the best compression ratio and is, therefore, used by default. If there are multiple runs of values though (Integer 2), combining NS & RLE yields better results.

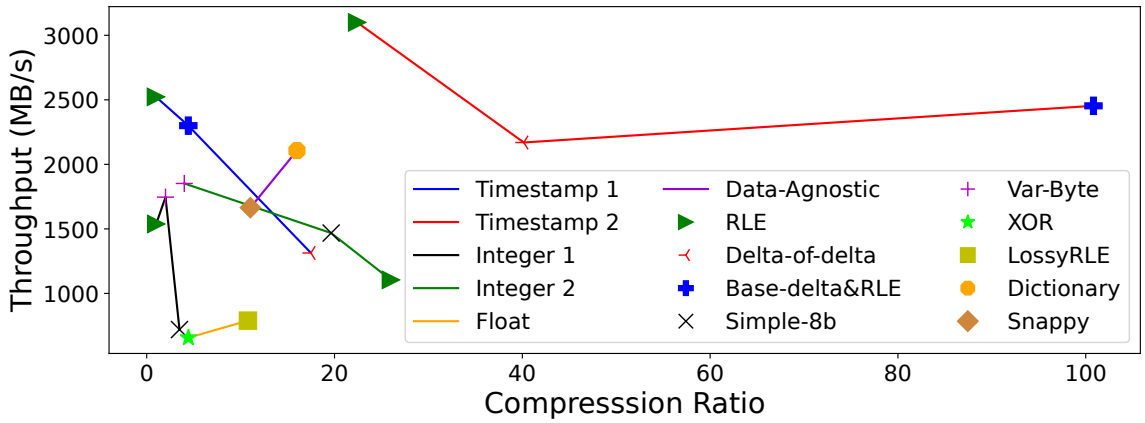


Fig. 5.7 Compression for various data types and distributions

Var-Byte has the highest throughput and is suitable for lower compression ratios when there is sufficient disk I/O bandwidth.

Floating-points. The nature of floating-point values makes them more challenging to compress efficiently with low overhead. XOR and lossy compression offer a good trade-off here. If full precision is unnecessary, the user can set the decimal point precision to a fixed error bound to improve compression. This lossy approach converts floating-point values into integers, allowing for integer compression schemes. In Figure 5.7, we use a floating-point stream with a predefined error bound, thus showing the performance difference between XOR and lossy compression.

Data-agnostic. For other data types in streams (e.g., fix-length strings), we observe, based on our evaluation, that dictionary compression works well, especially for a limited set of repeating values. When no statistics are available, Snappy is considered the default compression scheme. When it is possible to infer that the data can be mapped to a limited range of distinct values, it is better to use a static hashtable, shown as Dictionary in Figure 5.7.

Discussion. As shown in Figure 5.7, while some algorithms achieve the highest compression ratio, they have low throughput. The decision of the appropriate algorithm becomes even more complicated when considering that algorithms, such as lossy compression for floating-points or dictionary encoding, produce new data types that can be further compressed with other approaches. Therefore, there is a need to choose the compression algorithms adaptively.

5.4.3 Adaptive stream compression

The first step of adaptive stream compression involves profiling each pipeline fragment. For every pipeline fragment and input column in a p-stream, information is collected about the

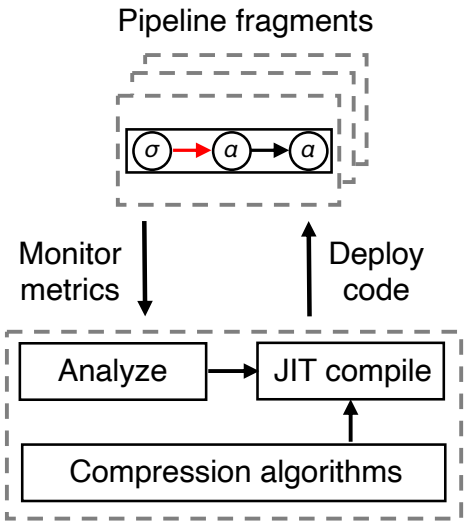


Fig. 5.8 Adaptive compression mechanism

value distribution (e.g., the min/max value) and characteristics specific to the compression schemes (e.g., the average run-length of consecutive equal values).

Following the metrics collection, analysis is performed using static information, e.g., the p-stream schema, and heuristics about the algorithms (i.e., we use a decision tree similar to previous work [1]). This analysis allows to reason about the data characteristics (e.g., data range) and choose the most appropriate scheme from a set of compression algorithms. Finally, code is generated for new compression operators and deployed back to the pipeline fragments of the POG.

At the beginning of query execution, each pipeline fragment starts with a predefined compression scheme per column (or without compression), as shown in line 13 of Figure 5.3. Upon detecting workload changes for a pipeline fragment, the adaptive optimization loop (shown in Figure 5.8): (i) JIT-compiles new compression and decompression operators; (ii) fuses them with the query-specific pruning operators from Section 5.3; and (iii) injects the generated function pointers into the respective pipeline fragments. In Section 5.5.3, we discuss the implementation details of the previous steps.

This adaptive approach supports a wide range of optimizations, such as selecting the most resource-efficient algorithm or specializing the underlying data structures (e.g., the hashtable for dictionary encoding). An example of such optimizations is using integers' bit precision to replace the more expensive Simple-8b algorithm with word-aligned NS; another example is using average run-length statistics to decide whether to use RLE.

5.5 System design and implementation

While POGs provide a high-level interface for fault-tolerance operations, an SPE must coordinate these operations efficiently, considering the limited single-node resources and workload characteristics. We describe SCABBARD, an SPE for multi-core CPUs that realizes the POG model and implements efficiently adaptive data compression. Its goal is to provide exactly-once fault-tolerance with minimal performance impact by making workload-aware decisions during execution. After an overview of the SCABBARD architecture (Section 5.5.1), we explain how SCABBARD manages persistent data, and reduces the recovery time (Section 5.5.2). Finally, we discuss how SCABBARD implements adaptive compression (Section 5.5.3).

5.5.1 SCABBARD system overview

SCABBARD is based on the query execution engine and compiler from LIGHTSABER [205]. To support persistence, SCABBARD introduces: (i) a *Block Manager* that stores streams and state; and (ii) a *Checkpoint Controller* that orchestrates consistent checkpoints and recovery. For efficient persistence, SCABBARD uses task-based parallelization for multi-core execution and adaptive data pruning for I/O bandwidth reduction. SCABBARD schedules tasks to a set of worker threads, with each worker bound to a physical CPU core. Depending on the number of pipeline breakers [234] (e.g., `AGGREGATION`), it instantiates one task dispatcher for each pipeline fragment when creating computational tasks.

In Figure 5.2, we introduced SCABBARD's architecture with a single operator pipeline, highlighting in red the features for workload-aware persistence. Next, we describe the different query execution stages, from the logical plan input to the generation of in-order complete window results.

In stage ❶, a user provides a stream query that is transformed into a logical plan. This plan is optimized in ❷ with rule-based optimizations including (i) operator reordering (i.e., persistence push-up) and (ii) operator fusion. SCABBARD uses the optimized plan to generate code for *persistence*, *checkpoint*, and *query* tasks.

The task creation stage ❸ follows after code generation. As data and markers arrive in the input queues of a query pipeline [165] through network sockets or RDMA, different tasks with their data dependencies are created and placed in system-wide queues based on NUMA data locality [229]. When the task queues contain tasks, the workers execute them in ❹.

To provide an up-to-date view of data characteristics (e.g., value distributions), the workers profile a subset of tuples before persistence in ❺. The profiling information may trigger another code generation process for workload-aware data reduction in step ❷ (see

Sections 5.4.3 and 5.5.3). Finally, the execution of a query task produces results in immutable batches, which are reordered and assembled in ❻ using the assembly function ρ^{α}.

5.5.2 Managing fault-tolerance operations

We now explain the role of SCABBARD's components, its data storage format, and how it accelerates persistence and recovery.

The **Block Manager** manages the persistent data of a query. When a p-stream or the Checkpoint Controller issue read/write requests to stable storage, they invoke the Block Manager, which returns a valid file pointer for these operations. The Block Manager maintains a pool of files, uniquely identified by a *FileId*, to reduce the overhead of OS file allocation. For p-streams (i.e., fixed-size queues based on the query configuration), the Block Manager maintains a circular list of files that maps directly to stream offsets because data is stored in offset order. The Block Manager also tracks which files must be garbage collected and returned to the pool.

The **Checkpoint Controller** coordinates persistence and recovery operations as discussed in Section 5.3.2. During normal execution, it injects markers to trigger the persistence of p-units and creates asynchronous tasks in stage ❸ for parallel execution. Task completion is monitored using a lock-free queue with atomics per pipeline to minimize overhead. When the Controller triggers a checkpoint, the operators withhold their outputs until checkpoint completion to ensure consistency. Regular processing is not disrupted, as the immutable p-units support persistence without an application-level copy-on-write operation.

Storage format. For data persistence, our goal is to perform parallel non-sequential disk operations without conflicts. Therefore, we partition each file into smaller logical segments (aligned 256 KB blocks), accelerating reads/writes at the expense of storage space [27].

Serialization costs are reduced by using state management primitives (e.g., vectors or hashtables) that contain tuples with primitive data types (e.g., integers) based on a fixed predefined schema. These primitive types do not require deserialization from storage without compression. For the retrieval of compressed data, however, metadata must be stored at the start of each segment: (i) the offsets of data; (ii) its representation (e.g., data types, row/-column format); and (iii) the used compression algorithms, which may change dynamically. For example, for dictionary encoding [189, 1], the hashtable's file offset (implemented with open addressing) and the metadata (e.g., schema) are stored to deserialize it. For windowed operators, the number and sizes of window fragments [205, 137] (e.g., open or closing windows) must be stored for state reconstruction.

I/O optimizations. SCABBARD supports NUMA-aware persistence: the task placement respects the affinity of p-units to reduce cross-socket communication. It also uses software prefetching of data from remote NUMA nodes, which leads up to 35% better performance for memory-bound queries. To saturate the I/O bandwidth of SSDs and minimize latency, it uses Linux's non-blocking API with asynchronous notifications [132]. All files are opened using the `O_DIRECT` flag to bypass the kernel's page cache and reduce the CPU overhead when performing I/O operations. Workers bulk up writes into chunks to decrease fragmentation and the number of entries in the disk's device queue.

To saturate the network bandwidth of RDMA-capable networks, SCABBARD uses the InfiniBand RDMA verb interface [73] with the two-sided communication primitives (see Section 2.2.3). Before initiating data processing, SCABBARD initializes a memory region for the circular buffers that act as input or output queues and guarantee FIFO delivery. Then, it registers these regions with the network card and establishes a connection with data sources or sinks. Every slot of these queues is a fixed-size (i.e., 1 MB), RDMA-capable buffer, reused after data ingestion or emission. SCABBARD performs network communication asynchronously to interleave network I/O with query execution.

Reducing recovery time. Fast recovery necessitates frequent checkpoints, short initialization times, and fast data loading from storage. SCABBARD reduces the checkpointing impact by performing them asynchronously. It also partitions streams and state into p-units to enable parallel persistence and recovery (see Sections 5.6.4 and 5.6.5). During recovery, SCABBARD avoids costly code generation by recovering previously-persisted compiled operators: it compiles queries using the LLVM compiler [142] and stores the binaries on disk. Upon restart, it loads the compiled operators, which reduces the restart time by an order of magnitude. Finally, dependency tracking allows SCABBARD to load only required p-units.

Making progress. As discussed above, SCABBARD adopts asynchronous execution using fixed-size lock-free queues protected by atomics for: (i) the assembly phase in ❻; (ii) stream and operator state persistence; and (iii) networking communication. To achieve high resource utilization, all tasks for the above operations finish in a finite number of steps without waiting to receive notifications for completeness. As this can block query execution for complex pipelines, the task dispatcher of every pipeline fragment tracks the size of these fixed-size queues and creates tasks that deque and resolve notifications. This *helping pattern* is similar to techniques found in concurrent programming to guarantee progress [103].

5.5.3 Efficient adaptive compression

Given the low latency requirement *R2* discussed at the beginning of this chapter, it is crucial to perform adaptive compression without disrupting normal execution. Therefore, SCABBARD generates query-specific lightweight instrumentation code and injects it into each pipeline fragment to perform fine-grained profiling. To further reduce the overhead of adaptive compression, we choose to analyze the statistics gathered at runtime periodically at a configurable interval[7] and decide whether to generate new operators. These operators are memoized and maintained as function pointers, inserted dynamically into the operator graph.

As the compression operators are stateless, the above approach may lead to different operators being executed simultaneously in SCABBARD. Therefore, workers must store the metadata of each approach (see Section 5.5.2) and use the appropriate generated decompression functions on the persisted p-units upon failure. If the p-stream characteristics change, e.g., a column's bit precision changes, a worker may decide for deoptimization [82, 100, 109] by falling back to the default compression scheme to ensure correct results without data loss. Following the deoptimization phase, the worker continues to use the default scheme (e.g., no compression if no operator is defined) until the next optimization interval of the adaptive mechanism generates a new operator.

Given the difference in size between p-streams and operator state, we decided to perform adaptive compression on p-streams as they have a greater impact on performance. To compress the operator state, we provide predefined compression functions in the current implementation. However, SCABBARD's compiler can be extended to enable adaptive compression for workloads that maintain large window state.

5.6 Evaluation

In this section, we evaluate SCABBARD to explore the benefits of its design in a top-down fashion over a range of synthetic and real-world datasets, which are presented in our evaluation set-up (Section 5.6.1). We start by comparing SCABBARD with state-of-the-art SPEs in terms of throughput and latency under a range of real-world query benchmarks (Section 5.6.2). We then investigate the efficiency of stream persistence against state-of-the-art messaging systems to reveal their overheads (Section 5.6.3). Subsequently, we demonstrate the efficiency of SCABBARD's checkpointing (Section 5.6.4) and recovery mechanisms (Section 5.6.5). Finally, we provide a breakdown of the optimization approaches (i.e., persistence

[7] It is statically defined, but it could change dynamically based on the collected statistics.

Table 5.3 Evaluation datasets and workloads

Datasets		Queries		
Name	# Attr. / Size (B)	Name	Windows (s)	Operators
Cluster Monitoring (CM) [129, 57]	12 / 64	CM_1	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{sum}}$
		CM_2	$\omega_{60,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$
Smart Grid (SG) [123]	7 / 32	SG_1	$\omega_{3600,1}$	π, α_{avg}
		SG_2	$\omega_{128,1}$	$\pi, \gamma, \alpha_{\text{avg}}$
		SG_3	$\omega_{1,1}, \omega_{1,1}$	$\pi, \sigma, \bowtie$
Linear Road Benchmark (LRB) [22]	7 / 32	LRB_1	$\omega_{300,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$
		LRB_2	$\omega_{30,1}$	$\pi, \gamma, \alpha_{\text{count}}$
		LRB_3	$\omega_{30,1}, \omega_{1,1}$	$\pi, \gamma, \alpha_{\text{count}}$
Yahoo Streaming (YSB) [58]	7 / 128	YSB	$\omega_{10,10}$	$\sigma, \pi, \bowtie_{\text{relation}}, \gamma, \alpha_{\text{count}}$
NEXMark (NQ) [214]	9 / 128	NQ	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{count}}, \alpha_{\text{max}}, \bowtie$
Sensor Monitoring (SM) [122]	14 / 64	SM	$\omega_{60,1}$	π, α_{avg}

push-up and compression) in Section 5.6.6 and analyze the performance of SCABBARD with remote sources, sinks, and storage in Section 5.6.7.

5.6.1 Experimental setup

Hardware. We run experiments on three servers: Server A with two Intel Xeon E5-2640 v3 2.60 GHz CPUs (16 physical cores), a 20 MB LLC cache, 64 GB of memory, and a local 256 GB SSD (950 MB/s write bandwidth; 72k IOPS); a c5.4xlarge AWS EC2 instance (Server B) with EBS [10] for remote storage (700 MB/s write bandwidth; 16k IOPS); Server C with four Intel Xeon E5-4660 v4 2.20 GHz (64 physical cores), a 40 MB LLC cache, 528 GB of memory, and a local 1.6 TB SSD (1.5 GB/s write bandwidth; 90k IOPS). We use Ubuntu 18.04 and Clang 9.0.0 with `-O3 -march=native`. Unless stated otherwise, all experiments are executed on Server A using all cores.

Stream persistence systems. To identify the performance characteristics of stream persistence, we compare to (i) Apache Kafka v2.3.0 [16], a persistent messaging system; and (ii) a C++ prototype (Kafka++) that flushes data to disk before acknowledging it.

For a fair comparison, we tune Kafka for high throughput by: (i) batching input messages; (ii) using multiple partitions and producers per topic; and (iii) maintaining a median latency of less than 90 ms, which we deem acceptable given the latency results measured below. We use `acks= "all"` mode to persist tuples to disk before acknowledging them, and

`replication.factor="1"`. We find that, in most cases, the compression algorithms supported by Kafka lead to performance degradation or increased latency; thus, we disable this feature. For the prototype, we manage memory and execution as in SCABBARD, pre-partition the input to avoid the additional cost, and use Snappy [241] compression.

Stream processing engines. For our end-to-end performance measurements, we compare to (i) Apache Flink v1.12.0 [15], a Java-based scale-out SPE; (ii) a hardcoded C++ implementation of Flink's execution strategy (Flink++); and (iii) LIGHTSABER [205], a single-node SPE without fault-tolerance.

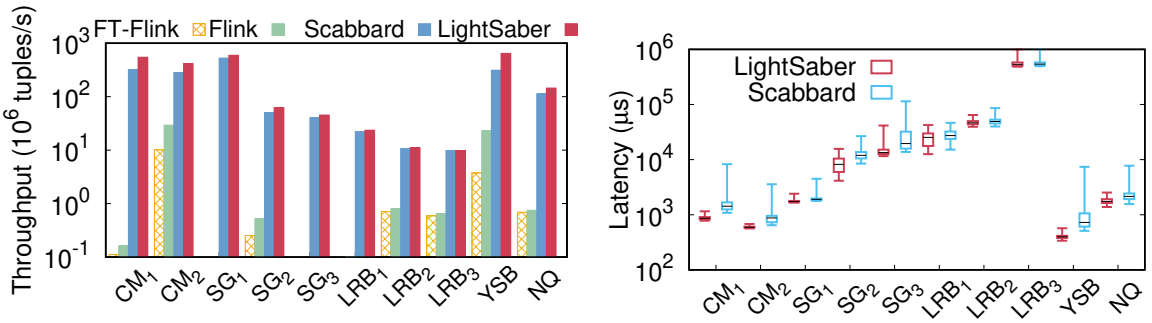
When considering a persistent input source in the following benchmarks, we use Kafka and Kafka++ for the Java-based and the C++ version of Flink, respectively. Following best practices [60] for data ingestion, we configure Kafka to use as many partitions as Flink workers. We enable object reuse and preload the input data into Kafka partitions before starting experiments to avoid bottlenecks. For Flink++, we pre-partition the input, perform operator fusion, and manage memory as in SCABBARD. We denote as Flink-Kafka++ the hardcoded C++ implementation that uses Kafka++.

We examine SCABBARD with and without stream persistence (denoted as Scabbard-Chk). If not stated otherwise, we checkpoint all operators every second and generate in-memory ingress streams for the remaining systems. We pre-populate large buffers and replay tuples continuously by updating their timestamps to avoid network bottlenecks.

Workloads. Table 5.3 summarizes the workloads used for our evaluation (see Section A.1 for their CQL definition), with the window sizes and slides measured in seconds[8] if not stated otherwise. We use the macro-benchmark stream queries from Section 4.7 and introduce four additional queries:

- The first workload emulates two **cluster monitoring** applications (CM) [220] that apply a grouped aggregation over a sliding window.
- **Smart grid** queries (SG) [123] perform anomaly detection: SG_1 calculates a sliding global average of a meter load, SG_2 reports the sliding load average per plug in a household, and SG_3 joins their results with a tumbling window.
- **Linear Road Benchmark** (LRB) [22] computes three queries on a network of toll roads with multiple key groupings: LRB_1 performs a grouped window aggregation with a selection to find congested road segments; LRB_2 and LRB_3 (a tumbling window count over LRB_2) count the number of vehicles in road segments.

[8] All window sizes and slides are defined using event time to be independent of processing latency.



(a) Performance for applications

(b) Latency for applications

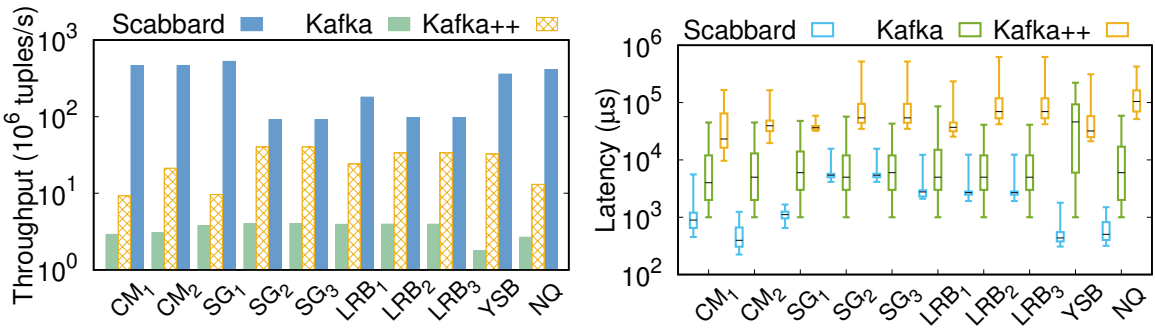
Fig. 5.9 Application benchmark queries

- **Yahoo Streaming Benchmark** (YSB) [58] emulates an advertisement application with a table join and a windowed count using numerical values (128 bits) [175].
- The fifth query (NQ) from **NEXMark benchmark** [214] that monitors auction items with the most bids over a sliding window.
- **Sensor monitoring** (SM) [122] query computes the running average of three energy sensor readings.

Metrics. The main performance metrics considered in the following benchmarks are throughput and end-to-end latency. We define throughput as the average number of tuples processed within a time unit (e.g., one second). Following prior work [217], we define end-to-end processing latency as the difference between the time when a tuple enters the system and when a window result is produced. Candlesticks in plots show the 5th, 25th, 50th, 75th and 95th percentiles, respectively.

5.6.2 System comparison using application benchmarks

To study how efficiently SCABBARD supports exactly-once fault-tolerance in a single node, we use ten queries from diverse streaming use cases with tumbling and sliding window semantics. Figure 5.9a compares the performance of Flink with 1-sec checkpoints (denoted as Flink-FT) with that of Flink without fault-tolerance, LIGHTSABER (no fault-tolerance), and SCABBARD. In particular, SCABBARD uses 1-sec checkpoints and persist its input streams. The results show that for compute-intensive queries (SG_3, $LRB_{1\text{-}3}$), SCABBARD exhibits less than 11% performance drop over LIGHTSABER, and effectively hides the cost of persistence. For the remaining memory-intensive queries, we observe that the overhead of persistence leads to a greater degradation: 69% for CM_1, 45% for CM_2, 12% for SG_1, 23% for SG_2, up to 2× for YSB, and 28% for NQ, respectively.



(a) Stream persistence results

(b) Latency for ingesting streams

Fig. 5.10 Ingestion of streams

Compared to Flink-FT, SCABBARD performs at least an order of magnitude better for all queries, even though it performs additional work for stream persistence. To investigate the fault-tolerance overhead, we use the bpftrace tools [98] and measure the average block I/O device latency for disk operations. While Flink has an average latency of 16 ms with frequent spikes (up to 64 ms), SCABBARD exhibits low and predictable (around 64 μs) average latency with 1 ms spikes by bypassing the kernel's page cache. The average disk latency explains the increased number of memory stalls for Flink that lead to a 4–6× performance overhead for $\mathsf{LRB}_{2\text{-}3}$ and YSB.

Next, we compare the end-to-end latency of SCABBARD against LIGHTSABER (we omit the results for Flink, as they are an order of magnitude worse [217, 99]). Figure 5.9b shows that, similar to LIGHTSABER, SCABBARD exhibits median latency lower than 50 ms for all queries, except for LRB_3, in which both systems have sub-second latency. For the compute-intensive queries (SG_3 and $\mathsf{LRB}_{1\text{-}3}$), the increase in latency is shown mostly in the 95$^{\text{th}}$ percentile, while for the rest, we observe that the median latency is more than 2× higher.

Discussion. Overall, the experiments show that SCABBARD achieves at least an order of magnitude higher throughput compared to state-of-the-art fault-tolerant SPEs, which reveals the benefits of our design in single-node deployments. Compared to efficient scale-up execution without fault-tolerance, SCABBARD achieves only an up to 10× increase in the 95$^{\text{th}}$ percentile latency and minimal throughput overhead due to its persistence abstractions and optimizations that reduce significantly disk bandwidth. Having established SCABBARD's high-level performance profile, we study the factors contributing to its performance.

5.6.3 Stream persistence cost

As a first step, we analyze SCABBARD's stream persistence mechanism against that of Kafka to reveal the overhead of existing approaches. Figure 5.10a shows that Kafka achieves

comparable performance for all applications (up to 4 million tuples/s). However, SCABBARD has at least two orders of magnitude greater throughput for all benchmarks. When analyzing resource utilization, we observe that Kafka introduces more instruction cache misses (the JVM leads to a large code footprint) and memory cache misses (caused by serialization, copying, and object allocation), which prevent scaling even with compression.

Subsequently, we remove the aforementioned bottlenecks with a hardcoded C++ implementation to provide a more comprehensive analysis. Kafka++ achieves up to an order of magnitude higher throughput and performs almost the same as SCABBARD for queries $SG_{2\text{-}3}$ and $LRB_{2\text{-}3}$, at the expense of a 7× latency increase, as shown in Figure 5.10b. This increase is caused by the large batch size required to achieve high throughput when performing synchronous disk writes.

In addition, we observe a significant percentage of stalls and high I/O device latency for both implementations that perform synchronous flushes to the page cache using `fsync()` (see Section 2.2.2). SCABBARD, in contrast, has more efficient resource utilization (e.g., NUMA locality), writes fewer bytes to disk per tuple, and reduces the transmission overhead with compression and block-aligned writes. Finally, it overlaps query execution with persistence and manages to submit more asynchronous I/O requests per second to disk.

Discussion. SCABBARD's persistence mechanism stresses less the device queues of SSDs and saturates their bandwidth with asynchronous I/O requests. By manually managing the memory for disk operations, as discussed in Section 2.2.2, and performing a set of optimizations (e.g., compression), SCABBARD achieves more than an order of magnitude better throughput or end-to-end latency compared to existing solutions.

Table 5.4 Checkpointing based on application characteristics

App	State (MB)		Avg checkpoint time (ms)		Overhead	
	SCABBARD	*Flink*	SCABBARD	*Flink*	SCABBARD	*Flink*
CM_1	18	0.08	25.7	292	4%	5%
CM_2	10	0.08	44.3	275	9%	3%
SG_1	2	0.03	89.6	291	1%	1%
SG_2	41	0.08	68.6	290	7%	11%
SG_3	115	3.3	103.6	$>$ 60000	2%	17%
LRB_1	114	4.2	122.7	9000	5%	15%
LRB_2	105	5.9	168.6	1000	14%	20%
LRB_3	143	6	361	2000	1%	10%
YSB	23	0.13	23.1	311	6%	1%
NQ	27	2.68	49.1	932	4%	10%

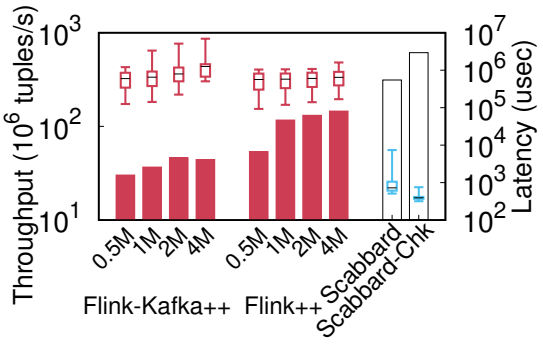


Fig. 5.11 Yahoo Streaming Benchmark

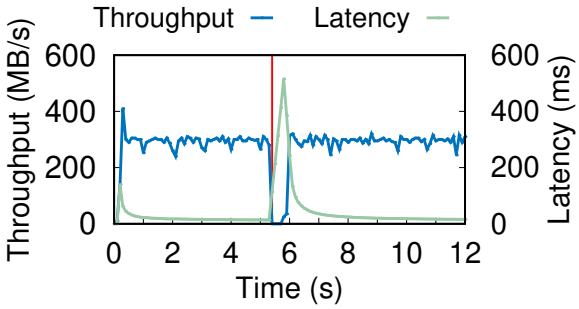


Fig. 5.12 Performance with failure for LRB_1

5.6.4 Checkpointing overhead

This section measures the performance overhead of checkpointing with a 1-sec interval for SCABBARD without p-streams (Scabbard-Chk) and Flink. Table 5.4 summarizes the results in terms of the average checkpoint size in MBs, the average checkpointing time in ms, and the performance overhead compared to execution without checkpoints.

First, we observe that for $\mathsf{LRB}_{1\text{-}3}$, the checkpoint time is affected by the persisted state size. Compared to Flink, SCABBARD has higher throughput for all queries, which leads to larger state sizes. When the state grows to several MBs, checkpointing affects performance adversely over time. Therefore, for queries SG_3 and $\mathsf{LRB}_{1\text{-}3}$, we had to increase Flink's checkpoint interval. Overall, SCABBARD combines efficient parallelization of persistence with data reduction and predictable I/O latency, allowing frequent snapshots and fast recovery.

We then consider the efficiency of unaligned checkpoints (i.e., persisting streams along with state) in a single-node SPE using Flink++ and YSB: we choose a workload with tumbling windows that allows a comparison without aggregation optimizations [205]. Flink++ uses aligned checkpointing in our implementation, i.e., persisting only state at the expense of increasing latency due to synchronization barriers.

Figure 5.11 compares Flink++ for different batch sizes with and without stream persistence (using Kafka++ from Section 5.6.3) to SCABBARD and Scabbard-Chk. With only checkpointing, the prototype exhibits 5× worse performance, two orders of magnitude higher latency, and 6× greater checkpoint time when the batch size is greater than 1 MB. This latency increase is due to message passing [234] and the alignment phase required during Flink's shuffle stage. While Flink++ waits for the checkpoint completion before committing results, SCABBARD uses its dependency tracking mechanism to output the results immediately. However, Flink++ stores to disk 100× less data with aligned checkpoints, demonstrating how the additional I/O pressure can become a bottleneck for SCABBARD with-

out the persistence optimizations. However, with stream persistence enabled, SCABBARD interleaves persistence with normal execution and yields 7× higher throughput.

Discussion. Overall, SCABBARD's checkpointing mechanism induces minimal performance overhead compared to execution without checkpoints. SCABBARD achieves lower average checkpoint time and latency than existing approaches for single-node fault-tolerance due to its asynchronous persistence and data reduction optimizations. Finally, we observe that unaligned checkpoints benefit the execution in a single node and reduce latency significantly.

5.6.5 Recovery with remote storage

In this experiment, we evaluate SCABBARD's behavior during recovery with remote storage, using an AWS EC2 instance with Elastic Block Storage (EBS). We exclude the time for failure detection and machine restart in our measurements. We use LRB_1 (the other queries exhibit similar behavior) and persist the ingress stream while checkpointing every second. To allow SCABBARD to catch up with the input when recovering, we configure the in-memory data generator to generate the stream at 300 MB/s (i.e., half the maximum sustainable throughput).

Figure 5.12 shows the throughput before and after manually triggering a failure by terminating the SCABBARD process, which is indicated by the red vertical line. Upon failure, SCABBARD initializes (memory pre-allocation and precompiled code loading), which takes approximately 360 ms, followed by 100 ms of recovery time. In terms of average latency, there is an initial increase while SCABBARD is down and restarts, but it then recovers to the pre-failure latency within 2 s.

To emulate a failure in Flink, we stop and restart the worker process (TaskManager) and collect the logged events. The restart time of the TaskManager is 38 s, and recovering the state from disk takes 2 s. This is the effective recovery time expected from a hot-standby system and, thus, the key metric of this experiment. Therefore, using this metric, SCABBARD performs roughly 20× better than Flink.

Discussion. SCABBARD achieves sub-second recovery even when using remote block storage. Its restart time is two orders of magnitude lower than existing solutions, making SCABBARD a practical solution for a single-node deployment. As future work, we want to automate the recovery process using efficient failure detection mechanisms and a pool of hot-standbys.

5.6.6 SCABBARD's optimization breakdown

We now study SCABBARD's data reduction techniques. In the first experiment, we execute the queries bound by the disk bandwidth (i.e., $\mathsf{CM}_{1\text{-}2}$, SG_1, YSB, NQ, SM) and evaluate

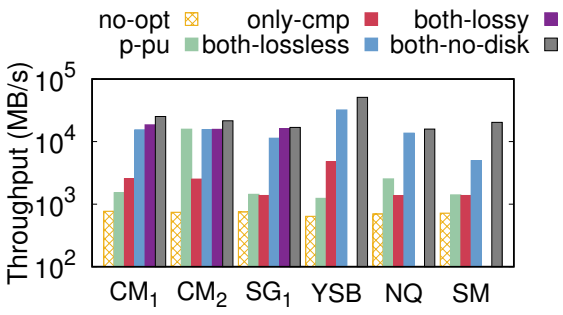


Fig. 5.13 Data reduction

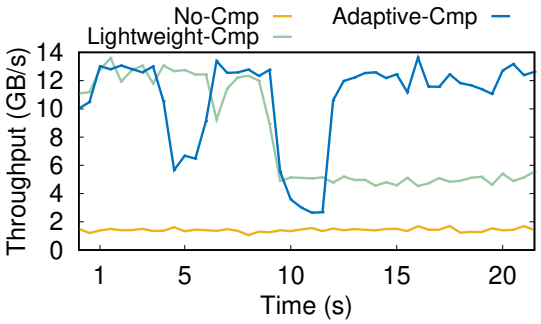


Fig. 5.14 Adaptive compression for SM

SCABBARD using six configurations: (i) no compression and no persistence push-up (no-opt); (ii) only persistence push-up (p-pu); (iii) only compression (only-cmp); (iv) both p-pu and lossless floating-point compression (both-lossless); (v) both p-pu and two-decimal digit precision floating-point compression if applicable (both-lossy); and (vi) both optimizations without persistence to emulate a fast storage medium (both-no-disk). The last configuration assumes that disk bandwidth is no longer a bottleneck in future hardware architectures (i.e., faster non-volatile memory).

Figure 5.13 shows that SCABBARD without data reduction reaches up to 780 MB/s, which is close to the disk bandwidth of Server A. For all queries apart from CM_2, which exhibits low filter selectivity, using only one of the techniques does not yield the optimal throughput. With both data reduction optimizations, SCABBARD outperforms the baseline (no-opt) from 7× to 50×, depending on the input data characteristics. For CM_1 and SG_1, the lossy compression yields 20–40% performance improvement.

We conclude that SCABBARD benefits from disks with higher bandwidth and lower latency operations, as we observe a 1.2–4× speedup for both-no-disk. With a faster disk, it may become necessary to sacrifice the compression ratio for throughput. We want to develop a cost-based model to resolve this as future work.

In the second experiment, we explore the performance benefit of adaptive compression when the data characteristics change over time. We execute the SM query, and after 10 secs, we change the value distribution of the integer columns.

Figure 5.14 compares SCABBARD with adaptive compression against the default compression approach and without compression. While SCABBARD profiles the runtime characteristics and decides to switch the compression function every 4 secs, this does not affect performance. Based on the collected statistics, SCABBARD generates a more efficient compression scheme, resulting in a 5–10% performance improvement. After 10 secs, the data characteristics change, invalidating the assumptions of the generated code, and SCABBARD falls back to the generic compression algorithms. Finally, after 12 secs, it uses the most

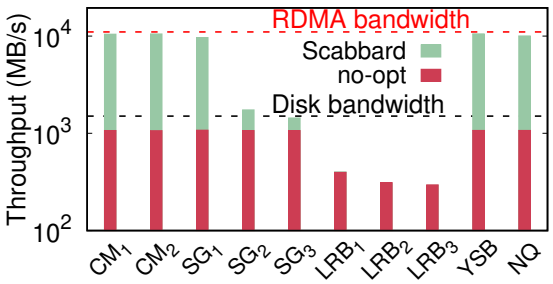


Fig. 5.15 Remote ingestion

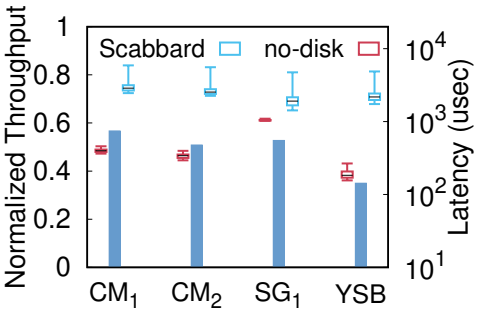


Fig. 5.16 Remote storage

recent statistics to generate a new compression function, yielding a 2× improvement. The re-optimization interval can be reduced to adapt more quickly at the cost of higher overhead. We deduce that SCABBARD adapts effectively to changing workload characteristics at runtime, resulting in up to a 2× performance gain.

Discussion. Overall, the experiments in Figures 5.13 and 5.14 show that both data reductions techniques improve the performance of memory-intensive queries with high ingestion rates up to an order of magnitude. In addition, adaptive compression can yield a 2× speedup for changing data characteristics without affecting normal execution.

5.6.7 Network and remote storage I/O bottlenecks

In the following experiments, we consider the impact of the network that interconnects SCABBARD with remote sources/sinks and storage. We observe its behavior when ingesting data over the network with and without data reduction (no-opt). To have sufficient bandwidth, we connect Server C (see Section 5.6.1) using RDMA over 100 Gb/s with two separate machines (similar to Server A) to generate streams and commit output results.

In Figure 5.15, for the memory-intensive queries, SCABBARD manages to saturate the RDMA bandwidth with less than 6 physical cores, which shows the importance of SCABBARD's data reduction techniques when the ingestion rate is higher than the disk bandwidth. The performance improvement for $SG_{2\text{-}3}$ is up to 65%, and for the remaining queries, data reduction does not improve performance and increases latency. This experiment reveals that data reduction plays a crucial role when utilizing fast networks.

In Figure 5.16, we compare SCABBARD's performance with and without (no-disk) remote block storage in terms of throughput and latency using the EC2 instance (Server B). For queries $CM_{1\text{-}2}$, SG_1, and YSB, SCABBARD exhibits greater throughput degradation compared to the local disk experiments (Section 5.6.2) because it saturates the IOPS of the EBS volume. Thus, we increase the batch size to reduce IOPS, which results in up to 12× higher 75th

percentile latency. The remaining queries exhibit similar performance to local storage with less than a 2× latency increase. We conclude that high-speed networking allows for remote storage with low overhead.

Discussion. In contrast to existing solutions [234], SCABBARD manages to saturate high-speed networking interconnects and allows for fast stream ingestion and remote storage. Therefore, its design is practical for many time-critical applications with strict end-to-end latency guarantees.

5.7 Limitations and discussion

In this section, we discuss related work, highlight some of SCABBARD's limitations, and initiate a discussion on how they can be addressed as future work.

POG assumptions and extensions. To guarantee exactly-once results upon failures, the persistence protocol described in Section 5.3.2 assumes that the data stream sources can buffer and replay data for a sub-second period (i.e., recovery interval). For stronger guarantees, the stream sources have to broadcast data to multiple replicated single-node deployments or a reliable distributed messaging system. Finally, the protocol assumes that the sink participates in the process to perform data deduplication.

Let us now discuss how to extend our persistence protocol for out-of-order data processing and non-deterministic operations. With respect to out-of-order data, SCABBARD can incorporate punctuation tuples [32] that act as markers for sorting tuples deterministically in a stream. For the support of non-deterministic operations, the protocol must log all non-deterministic decisions and replay them for recovery [219, 6]: input and output channels must be replaced with p-streams for logging all tuples [75], which may incur a high overhead. New operators can be specified as user-defined functions (UDFs) by implementing the interface from Table 5.1, while a similar approach to [195] can be used to capture all the sources of non-determinism.

Fault-tolerance and data migration mechanisms. This work addressed the challenge of designing a single-node fault-tolerant SPE with exactly-once semantics. Compared to systems with partial fault-tolerance [112, 119] that sacrifice the precision of recovered results, SCABBARD offers stronger processing guarantees. More recent scale-out systems [213, 6, 80, 120, 44, 44] use checkpointing for fault-tolerance: IBM Streams, Apache Flink, and Naiad employ a variation of the Chandy-Lamport algorithm [52] but are not designed for persisting streams efficiently. Instead, these systems rely on messaging systems [16, 11, 180] and general-purpose stores [78, 50]. In contrast, SCABBARD integrates persistence with the operator graph to enable workload-aware optimizations. Another common approach is the use

of a lineage-based mechanism [24, 149, 181] that persists all data dependencies, which would compromise performance for scale-up designs. To achieve high availability with minimal downtime after a failure, SCABBARD can be combined with active replication [112, 48]. Finally, data migration (e.g., Rhino [161] or Megaphone [108]) is an orthogonal technique that uses fast remote storage to speed up recovery to a new machine, and it also enables query reconfiguration at runtime.

Adaptive optimizations in stream processing. Adaptive techniques have been used extensively in SPEs [28, 55, 240, 88, 137, 100]. Early research focused on plan migration [55, 240, 88] in distributed deployments or operator reordering [28]. SABER [137] uses an online algorithm to choose between CPU and GPU execution of operators. Grizzly [100] employs adaptive optimizations with query compilation to accelerate execution. These approaches are orthogonal to our work, which uses adaptive compression to reduce I/O bandwidth, and we want to incorporate them into our design for improved performance.

Compression algorithms. In this work, we consider only a subset of existing approaches for lightweight compression, and we want to explore more schemes for future work. In particular, we utilize compression schemes for stream timestamps and floats [173] and variations of null-suppression methods [189, 1, 13, 145]. While SCABBARD uses compression to accelerate persistence, Tersecades [171] compresses data with hardware accelerators and performs computations directly over compressed formats. A noteworthy future direction is the extension of the adaptive compression and execution model to support such operations.

5.8 Summary

To enable fault-tolerance with exactly-once semantics in a single-node SPE without compromising performance despite the limited available resources, we developed SCABBARD. It tightly couples the persistence operations with the operator graph through a novel *persistent operator graph* model and dynamically reduces the required disk bandwidth at runtime through adaptive compression. SCABBARD achieves sub-second recovery latencies by performing frequent checkpointing and optimistic garbage collection. Consequently, it outperforms the state-of-the-art fault-tolerant SPEs by at least an order of magnitude on all our benchmarks, processing hundreds of millions of tuples/s with millisecond latencies.

We consider SCABBARD a comprehensive design for hardware-efficient, reliable, and general-purpose scale-up SPE. SCABBARD exploits existing hardware in terms of multi-core execution, network, and storage stack and provides a practical solution for real-world use cases. Therefore, it is a step towards next-generation SPEs that can replace cluster-based designs' complexity and operational costs with single-node deployments.

Chapter 6

Conclusions and Future Work

Over the past decade, we have witnessed unprecedented magnitudes of data volumes and velocity that demand real-time processing. Based on recent predictions [186], by 2025, data will reach 175 ZB with 30% being analyzed in real-time. Therefore, it is not surprising that stream processing has become the fourth most important data-intensive application workload. The stream processing paradigm is extensively used in a wide variety of domains ranging from finance [79, 197, 166] to transportation [22], healthcare [235], and e-commerce services [95, 49, 6]. With the continuously growing data volumes, velocity, and performance requirements (i.e., high throughput and low latency), there is a need to revisit the design of SPEs based on current hardware trends (e.g., non-uniform memory access, multi-core parallelism, and high-speed networks).

To accommodate the requirements of modern stream processing applications, many distributed SPEs were developed by large internet companies [163, 7, 6, 120, 49, 47, 211] and academia [80, 8] to scale out processing to a cluster of nodes through appropriate data partitioning [45, 231] – at substantial operational cost [155]. At the same time, with the rise of parallel hardware, such as multi-core CPUs and GPUs, and modern network technologies [64], we observe that scale-up SPE designs [137, 158, 236, 205, 157] have become a practical alternative. In particular, for a class of *time-critical* applications [235, 166], scale-up designs achieve higher performance and more predictable latency using fewer resources [175]. However, existing single-node systems neglect tolerating failures and implement ad-hoc aggregation and parallelization strategies, yielding high performance only for specific workloads.

With this thesis, we propose a novel *general-purpose* relational SPE that transparently exploits modern hardware trends to permit streaming *windowed* [21] SQL queries. Our solution achieves performance desiderata of stateful streaming applications regardless of the workload characteristics (e.g., number of distinct keys or window semantics). To address

the reliability challenge, we introduce single-node fault-tolerance mechanisms that provide predictable low latency results upon failures. Therefore, we present a practical single-node design for adoption in real-world use cases, which reduces the resource and maintenance footprint of stream processing systems.

Single-node SPEs provide an effective solution for scalable and reliable stream processing without compromising window semantics, performance requirements, or fault-tolerance.

In the remainder of this chapter, we summarize the contributions of this thesis and then discuss potential future work.

6.1 Thesis summary

This thesis began by describing the importance of big data analysis to various domains of our everyday lives (Chapter 1). With the recent technological advancements, the bottleneck of analyzing big data has shifted from generating and storing data to processing systems. Therefore, big internet companies and organizations have focused on efficiently processing these ever-growing data volumes to extract valuable information and gain insights.

While the classical "process-after-store" model was sufficient for processing increasing data volumes, a new type of latency-sensitive applications has gained attention in the last decades, posing a velocity challenge. These applications continuously produce "fresh" results as new data streams arrive. Given the predicted data volumes and velocities, high throughput and low latency performance are key requirements for stream processing.

To target these requirements, existing SPEs scale out data processing on commodity clusters, paying a substantial operational and maintenance cost. However, such designs face challenges in providing predictable latency guarantees and parallelizing window computations. Witnessing the emergence of shared-memory multi-core CPU architectures and high-speed networking, single-node SPEs have become a reasonable alternative.

In the background chapter (Chapter 2), we described the basic concepts and terminology for stream processing. Next, we traced the evolution of SPEs from single-core research prototypes to distributed stream processing, which has become a first-class service in cloud vendors nowadays. While current SPEs focus on the inherent problems of distributed execution, a wave of hardware-conscious systems has emerged to exploit parallel hardware (e.g., multi-core CPUs, GPUs, or FPGAs). However, existing solutions lack a comprehensive design for scalable and reliable single-node execution. After presenting the SPEs' evolution, we introduced the characteristics of modern hardware found in data centers, essential for profiling and designing efficient processing systems. We cover basic knowledge related to modern processors (i.e., microarchitecture), storage (i.e., asynchronous direct I/O), and

the network stack (RDMA capable networks). Subsequently, we focused on the single-core execution of window aggregation, a key component for streaming analytics. We then identified the limitations of existing solutions for overlapping windows caused by sliding semantics or multi-query execution. Following the challenges of efficient single-core execution, we track the well-known limitations of current parallelization strategies for window operations by analyzing different workloads. Finally, we examined existing fault-tolerant streaming approaches and demonstrated their limitations for single-node deployments due to resource constraints with a range of real-world applications.

To overcome the shortcomings related to single-core window aggregation, we presented two techniques that accelerate *computation sharing* for overlapping windows (Chapter 3). As a first step, we performed an in-depth analysis of existing approaches to identify the performance patterns of streaming algorithms. Based on this analysis, we proposed a solution that addresses two individual challenges: (i) accelerating partial window aggregation despite the complex data dependencies of overlapping windows; and (ii) performing incremental execution over multiple concurrent queries efficiently. For the first problem, we contributed HammerSlide that applies hardware-conscious optimizations (i.e., SIMD intrinsics) for partial aggregation and achieves up to 12× performance improvement when integrated with an SPE. For multi-query execution, we introduced SlideSide, an incremental algorithm that increases intermediate result sharing by exploiting the algebraic properties of aggregate functions. SlideSide achieves up to 2× better throughput and comparable latency against state-of-the-art incremental algorithms.

In Chapter 4, we address the challenge of parallelizing window aggregation without compromising semantics on multi-core CPUs with LIGHTSABER, a novel SPE that balances parallelism and incremental execution. In particular, we introduced a general aggregation model that captures existing design decisions and enables reasoning about entirely new ones. Based on this model, we introduced abstractions that generalize existing approaches for both parallel and incremental processing. For parallel aggregation, LIGHTSABER constructs an *aggregation tree* abstraction that exploits the parallelism of modern processors: it divides computation into intermediate steps that enable data- and task-level parallelism. Next, we focused on the design of a component that generates code for streaming applications at a query level (e.g., operator fusion) and incremental execution level. For the latter case, we introduced a second abstraction called *generalized aggregation graph* (GAG) that encodes the low-level data dependencies required to produce aggregates incrementally over a stream. This graph generalizes state-of-the-art incremental algorithms and adapts to workload characteristics (e.g., algebraic properties or number of queries). Before evaluating LIGHTSABER, we discussed how to perform memory management and NUMA-aware execution. LIGHTSABER

processes 470 million tuples/s with 132 μs median latency on a 16-core server. At the same time, it outperforms state-of-the-art systems, such as Apache Flink, by at least a factor of seven over a range of workloads. Our system is publicly available as open-source.

Then, in Chapter 5, we showed how we can implement single-node fault-tolerance without compromising performance as part of SCABBARD, a novel SPE that reduces the required disk I/O bandwidth while providing exactly-once results upon failures. To achieve this, we introduced a novel *persistent operator graph* model that enables *workload-aware* decisions for data persistence: persist streams based on the operators' selectivity. To further reduce the persisted data, we designed a mechanism for query-specific adaptive compression. SCABBARD uses JIT code generation to select suitable compression algorithms based on stream statistics collected at runtime. By combining both approaches, SCABBARD induces minimal overhead and recovers with sub-second latencies from failures. Therefore, it achieves processing throughput of 200 million tuples/s while outperforming existing systems, such as Apache Flink, by at least one order of magnitude. It is publicly available as open-source.

6.2 Future work

During the writing of this thesis, we have identified several topics for future work that we will discuss next.

Distributed stream processing. In this work, we focus on designing scalable and reliable SPEs for single-node execution based on the assumption that existing streaming workloads fit in shared-memory multi-core architectures. However, the appearance of applications with more complex requirements (e.g., maintaining machine learning models), along with the increase in data volumes and velocity, stress single-node resources for real-time analytics. Therefore, extending our abstractions for distributed execution models would be an interesting future direction. For example, as discussed in Section 4.3, our generalized aggregation model allows us to express novel distributed execution strategies. To avoid the expensive partitioning step incorporated by modern SPEs, such a strategy would perform computations locally in each node, similar to LIGHTSABER, before merging results in a distributed fashion. The distributed merge phase is required for operators that materialize their results (e.g., pipeline breakers) and can be accelerated by exploiting high-speed network interconnects and distributed consistency protocols.

In the context of distributed execution, our design would also have to revisit its execution model to support the parallelization strategy discussed above. As described in Section 2.1.2, in this thesis, we adopt a variation of the micro-batch approach to control task placement, which results in comparable end-to-end latency and higher throughput compared to materialized

tasks. It is interesting, thus, to investigate whether the task placement model is better suited for a distributed parallelization strategy.

Elastic execution. As a step further, our solution could also account for elasticity [80], i.e., scale computational resources dynamically depending on resource demand. This is an important property for applications running on public clouds, preventing overprovisioning of computing resources and cutting operational costs. Given the unpredictable load peaks of streaming workloads, providing cost-efficient elastic scaling for SPEs without compromising the quality of service becomes a major challenge.

Handle stream imperfections. In this thesis, the proposed solutions consider "in-order" or "slightly out-of-order" tuples from a single ingestion point, which can be buffered and reordered before applying any computations. However, in many scenarios, tuples from data streams are missing or arrive out-of-order [158] (i.e., late, out-of-sequence, or from multiple ingestion points). Therefore, SPEs must account for such imperfections [197, 9] without blocking execution. For missing tuples, our solution could apply lightweight imputation methods (i.e., filling missing values), similar to previous approaches in relational databases [160]. For out-of-order tuples, we can extend our processing and fault-tolerance models by using punctuation tuples [32] or low watermarks [6] to sort tuples deterministically within a stream and provide correctness guarantees.

Unify streaming and historical data analytics. While in our work, we consider only window computations over most recent data, many streaming applications require access to previously stored state [197]. With SCABBARD, we propose a storage management solution for stream and state persistence that leverages modern hardware. However, our proposal neglects open questions regarding storage formats [124], indexing techniques [223] or transactional semantics [156] for streaming applications. We leave for future work the extension of our storage layer to support analytics over both streaming and stored data.

Hardware accelerators. Even though we proposed an execution model that benefits from shared-memory multi-core CPU architectures, single-node performance can be enhanced by heterogeneous architectures [238, 208, 137] for different operations, such as query execution and cross-operator communication. Prior approaches [137, 239] have utilized GPUs for streaming operators, as they provide much higher bandwidth than CPUs. Glacier [162] is an engine that applies streaming computations on data from networks using FPGAs. Finally, we could extend SCABBARD to compress data using hardware accelerators [171], such as GPUs, and apply computations directly over compressed data formats.

Multi-tenant execution. As many streaming applications involve simple transformations over data streams with low input rates, there is an opportunity to exploit all available hardware resources by supporting multi-tenancy, i.e., multiple queries collocated on shared

resources. However, it is challenging to design a multi-tenant SPE that maintains high resource utilization without overprovisioning [225]. As a future direction, we could integrate resource managers with application-level metrics, such as load spikes, to dynamically change resource allocation among tenants.

References

[1] Abadi, D., Madden, S., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD*, pages 671–682.

[2] Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139.

[3] Abadi, D. J., Madden, S. R., and Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? In *ACM SIGMOD*, pages 967–980.

[4] 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: May 17, 2023.

[5] Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., et al. (2005). Distributed operation in the borealis stream processing engine. In *SIGMOD*, pages 882–884.

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

[7] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. (2015). The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *Proc. VLDB Endow.*, volume 8, pages 1792–1803.

[8] Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., et al. (2014). The stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964.

[9] Ali, M., Chandramouli, B., Goldstein, J., and Schindlauer, R. (2011). The extensibility framework in microsoft streaminsight. In *ICDE*, pages 1242–1253.

[10] Amazon (2022a). Amazon elastic block store. https://aws.amazon.com/ebs/. Last access: May 17, 2023.

[11] Amazon (2022b). Amazon Kinesis. https://aws.amazon.com/kinesis/data-streams/. Last access: May 17, 2023.

[12] AMD (2022). AMD64 Architecture Programmer's Manual Volume 2: System Programming.

[13] Anh, V. N. and Moffat, A. (2010). Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147.

[14] Apache ActiveMQ (2022). https://activemq.apache.org. Last access: May 17, 2023.

[15] Apache Flink (2022). https://flink.apache.org. Last access: May 17, 2023.

[16] Apache Kafka (2022). https://kafka.apache.org/. Last access: May 17, 2023.

[17] Apache Spark (2022). https://spark.apache.org. Last access: May 17, 2023.

[18] Apache Thrift (2022). https://thrift.apache.org. Last access: May 17, 2023.

[19] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2016). Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer.

[20] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). Stream: the stanford stream data manager (demonstration description). In *SIGMOD*, pages 665–665.

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

[22] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear road: A stream data management benchmark. In *Proc. VLDB Endow.*, volume 30, page 480–491.

[23] Arasu, A. and Widom, J. (2004). Resource sharing in continuous sliding-window aggregates. In *Proc. VLDB Endow.*, pages 336–347.

[24] Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., and Zaharia, M. (2018). Structured streaming: A declarative api for real-time applications in apache spark. In *ACM SIGMOD*, page 601–613.

[25] Association, I. T. (2007). InfiniBand Architecture Specification, Volume 1, Release 1.2.1.

[26] Association, I. T. (2022). Infiniband roadmap - advancing infiniband. https://www.infinibandta.org/infiniband-roadmap/. Last access: May 17, 2023.

[27] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., and Callaghan, M. (2016). Designing access methods: The rum conjecture. In *EDBT*, pages 461–466.

[28] Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272.

[29] Azure (2022). iot central. https://azure.microsoft.com/en-us/services/iot-central/. Last access: May 17, 2023.

[30] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proc. of SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16.

[31] Babu, S. and Bizarro, P. (2005). Adaptive query processing in the looking glass. In *CIDR*.

[32] Balazinska, M. (2005). *Fault-tolerance and load management in a distributed stream processing system*. PhD thesis, Massachusetts Institute of Technology.

[33] Balkesen, C. and Tatbul, N. (2011). Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In *DMSN*.

[34] Bancilhon, F., Briggs, T., Khoshafian, S., and Valduriez, P. (1987). Fad, a powerful and simple database language. In *VLDB*, volume 87, pages 1–4.

[35] Barak, D. (2014). Introduction to Remote Direct Memory Access (RDMA). https://www.rdmamojo.com/2014/03/31/remote-direct-memory-access-rdma/. Last access: May 17, 2023.

[36] Barthels, C., Loesing, S., Alonso, G., and Kossmann, D. (2015). Rack-scale in-memory join processing using RDMA. In *ACM SIGMOD*, pages 1463–1475.

[37] Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest? (2020). https://www.confluent.io/blog/kafka-fastest-messaging-system/. Last access: May 17, 2023.

[38] Bhatotia, P., Acar, U. A., Junqueira, F. P., and Rodrigues, R. (2014). Slider: Incremental sliding window analytics. In *Middleware*, pages 61–72.

[39] Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H., and Moran, C. (2010). Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *ACM SIGMOD*, pages 1093–1104.

[40] Bingman, B. (2018). Poor performance with sliding time windows. in flink jira issues. https://issues.apache.org/jira/browse/FLINK-6990. Last access: May 17, 2023.

[41] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., and Zamanian, E. (2016). The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539.

[42] Blelloch, G. E. (1990). *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA.

[43] Boncz, P. A., Zukowski, M., and Nes, N. (2005). Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237.

[44] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. (2017). State management in apache flink®: consistent stateful distributed stream processing. volume 10, pages 1718–1729.

[45] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *IEEE TCDE*, 36(4).

[46] Carbone, P., Traub, J., Katsifodimos, A., Haridi, S., and Markl, V. (2016). Cutty: Aggregate sharing for user-defined windows. In *CIKM*, pages 1201–1210.

[47] Chandramouli, B., Fernandez, R. C., Goldstein, J., Eldawy, A., and Quamar, A. (2016). Quill: Efficient, transferable, and rich analytics at scale. *Proc. VLDB Endow.*, 9(14):1623–1634.

[48] Chandramouli, B. and Goldstein, J. (2017). Shrink: Prescribing resiliency solutions for streaming. *Proc. VLDB Endow.*, 10(5):505–516.

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

[50] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. (2018). Faster: A concurrent key-value store with in-place updates. In *ACM SIGMOD*, pages 275–290.

[51] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668.

[52] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM TOCS*, pages 63–75.

[53] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM TOCS*, pages 1–26.

[54] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. (2016). Realtime data processing at facebook. In *SIGMOD*, pages 1087–1098.

[55] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000a). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD Rec.*, pages 379–390.

[56] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000b). Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390.

[57] Chen, X., Lu, C.-D., and Pattabiraman, K. (2014). Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *ISSRE*, pages 167–177.

[58] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B. J., and Poulosky, P. (2016). Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IEEE IPDPSW*, pages 1789–1792.

[59] Cieslewicz, J. and Ross, K. A. (2007). Adaptive aggregation on chip multiprocessors. In *Proc. VLDB Endow.*, page 339–350.

[60] Confluent (2020). Optimizing your apache kafka deployment. https://www.confluent.io/thank-you/white-paper/optimizing-your-apache-kafka-deployment/. Last access: May 17, 2023.

[61] Consortium, O. (2017a). Smart cities scenario. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. Last access: May 17, 2023.

[62] Consortium, O. (2017b). Transportation scenario: Smart cars and traffic control. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. Last access: May 17, 2023.

[63] Corporation, M., Corporation, I., and Corporation, C. C. (1997). Virtual interface architecture specification.

[64] Crago, S., Dunn, K., Eads, P., Hochstein, L., Kang, D.-I., Kang, M., Modium, D., Singh, K., Suh, J., and Walters, J. P. (2011). Heterogeneous cloud computing. In *ICCC*, pages 378–385.

[65] Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *CSUR*, 44(3):1–62.

[66] Cutting, D. and Pedersen, J. (1989). Optimization for dynamic inverted index maintenance. In *ACM SIGIR*, page 405–411.

[67] Damme, P., Ungethüm, A., Hildebrandt, J., Habich, D., and Lehner, W. (2019). From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM TODS*, pages 1–46.

[68] De Matteis, T. and Mencagli, G. (2017). Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *International Journal of Parallel Programming*, 45(2):382–401.

[69] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[70] Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

[71] Dobbelaere, P. and Esmaili, K. S. (2017). Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *ACM DEBS*, pages 227–238.

[72] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. (2017). Optimizing space amplification in rocksdb. In *CIDR*.

[73] Dragojevic, A., Narayanan, D., and Castro, M. (2017). Rdma reads: To use or not to use? *IEEE Data Eng. Bull.*, 40:3–14.

[74] Elmqvist, N. and Fekete, J.-D. (2009). Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16:439–454.

[75] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM CSUR*, pages 375–408.

[76] Esper (2023). http://www.espertech.com/esper/. Last access: May 17, 2023.

[77] Estan, C. and Varghese, G. (2002). New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336.

[78] Facebook (2012). RocksDB. http://rocksdb.org/. Last access: May 17, 2023.

[79] feedzai.com (2013). Modern Payment Fraud Prevention at Big Data Scale. http://tinyurl.com/nwnzdxs.

[80] Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *ACM SIGMOD*, pages 725–736.

[81] Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2014). Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, pages 49–60.

[82] Fink, S. J. and Qian, F. (2003). Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO*, pages 241–252.

[83] Flajolet, P. and Martin, G. N. (1983). Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 76–82. IEEE.

[84] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960.

[85] Fragkoulis, M., Carbone, P., Kalavri, V., and Katsifodimos, A. (2020). A survey on the evolution of stream processing systems. *arXiv preprint arXiv:2008.00842*.

[86] Garefalakis, P. (2020). *Supporting long-running applications in shared compute clusters*. PhD thesis, Imperial College London.

[87] Garefalakis, P., Karanasos, K., and Pietzuch, P. (2019). Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *SoCC*, pages 233–245.

[88] Gedik, B., Andrade, H., and Wu, K.-L. (2009a). A code generation approach to optimizing high-performance distributed data stream processing. In *ACM SIGMOD*, pages 847–856.

[89] Gedik, B., Bordawekar, R. R., and Philip, S. Y. (2009b). Celljoin: a parallel stream join operator for the cell processor. *The VLDB journal*, 18(2):501–519.

[90] Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2001). Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, pages 79–88.

[91] Gill, P., Jain, N., and Nagappan, N. (2011). Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, page 350–361.

[92] Golab, L. and Özsu, M. T. (2003). Issues in data stream management. *Sigmod Record*, 32:5–14.

[93] Google (2022). Google compute engine persistent disk. https://cloud.google.com/persistent-disk. Last access: May 17, 2023.

[94] Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006). Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336.

[95] Graepel, T., Candela, J. Q., Borchert, T., and Herbrich, R. (2010). Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *ICML*, pages 13–20.

[96] Grafana (2022). http://grafana.org/. Last access: May 17, 2023.

[97] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53.

[98] Gregg, B. (2019). *BPF Performance Tools*. Addison-Wesley Professional.

[99] Grier, J. (2016). Extending the yahoo! streaming benchmark. https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark. Last access: May 17, 2023.

[100] Grulich, P. M., Sebastian, B., Zeuch, S., Traub, J., Bleichert, J. v., Chen, Z., Rabl, T., and Markl, V. (2020). Grizzly: Efficient stream processing through adaptive query compilation. In *ACM SIGMOD*, pages 2487–2503.

[101] Han, S., Egi, N., Panda, A., Ratnasamy, S., Shi, G., and Shenker, S. (2013). Network support for resource disaggregation in next-generation datacenters. In *ACM HotNets*, pages 1–7.

[102] Héman, S., Nes, N., Zukowski, M., and Boncz, P. (2007). Vectorized data processing on the cell broadband engine. In *DaMoN*, pages 1–6.

[103] Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.

[104] Hettinger, R. (2009). Efficient running median using an indexable skiplist (python recipe). https://code.activestate.com/recipes/576930/. Last access: May 17, 2023.

[105] Hirzel, M., Rabbah, R., Suter, P., Tardieu, O., and Vaziri, M. (2016). Spreadsheets for stream processing with unbounded windows and partitions. In *DEBS*, pages 49–60.

[106] Hirzel, M., Schneider, S., and Tangwongsan, K. (2017). Sliding-window aggregation algorithms: Tutorial. In *DEBS*, pages 11–14.

[107] Hirzel, M., Soulé, R., Schneider, S., Gedik, B., and Grimm, R. (2014). A catalog of stream processing optimizations. *CSUR*, 46(4):1–34.

[108] Hoffmann, M., Lattuada, A., McSherry, F., Kalavri, V., Liagouris, J., and Roscoe, T. (2019). Megaphone: Latency-conscious state migration for distributed streaming dataflows. In *Proc. VLDB Endow.*, volume 12, pages 1002–1015.

[109] Hölzle, U., Chambers, C., and Ungar, D. (1992). Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN*, pages 32–43.

[110] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *IRE*, pages 1098–1101.

[111] Hung, M. (2017). Leading the iot, gartner insights on how to lead in a connected world. https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf. Last access: May 17, 2023.

[112] Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. (2005). High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790.

[113] IEEE and Group, T. O. (2007). The open group base specifications issue 7, 2018 edition.

[114] Intel (2009). An Introduction to the Intel® QuickPath Interconnect.

[115] Intel (2010). Understanding iwarp: Delivering low latency to ethernet.

[116] Intel (2021). Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.

[117] Intel (2023). Intel threading building blocks. https://software.intel.com/en-us/intel-tbb. Last access: May 17, 2023.

[118] Introduction to Kafka Streams (2017). http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple. Last access: May 17, 2023.

[119] Jacques-Silva, G., Gedik, B., Andrade, H., and Wu, K.-L. (2009). Language level checkpointing support for stream processing applications. In *DSN*, pages 145–154.

[120] Jacques-Silva, G., Zheng, F., Debrunner, D., Wu, K.-L., Dogaru, V., Johnson, E., Spicer, M., and Sariyüce, A. E. (2016). Consistent regions: Guaranteed tuple processing in ibm streams. In *Proc. VLDB Endow.*, volume 9, pages 1341–1352.

[121] Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. (2006). Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD*, pages 431–442.

[122] Jerzak, Z., Heinze, T., Fehr, M., Gröber, D., Hartung, R., and Stojanovic, N. (2012). The debs 2012 grand challenge. In *ACM DEBS*, pages 393–398.

[123] Jerzak, Z. and Ziekow, H. (2014). The DEBS 2014 Grand Challenge. In *ACM DEBS*, pages 266–269.

[124] Kalavri, V. and Liagouris, J. (2020). In support of workload-aware streaming state management. In *USENIX HotStorage*.

[125] Kalia, A., Kaminsky, M., and Andersen, D. G. (2016). Design guidelines for high performance rdma systems. In *USENIX ATC*, pages 437–450.

[126] Kang, J., Naughton, J. F., and Viglas, S. D. (2003). Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352.

[127] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., and Markl, V. (2018). Benchmarking distributed stream data processing systems. In *ICDE*, pages 1507–1518.

[128] Karnagel, T., Habich, D., Schlegel, B., and Lehner, W. (2013). The hells-join: a heterogeneous stream join for extremely large windows. In *DaMoN*, pages 1–7.

[129] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. (2010). An Analysis of Traces from a Production MapReduce Cluster. In *CCGrid*, pages 94–103.

[130] Kazemitabar, S. J., Demiryurek, U., Ali, M., Akdogan, A., and Shahabi, C. (2010). Geospatial Stream Query Processing Using Microsoft SQL Server StreamInsight. *Proc. VLDB Endow.*, 3:1537–1540.

[131] Kemper, A. and Neumann, T. (2011). Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE.

[132] Kernel Asynchronous I/O for Linux (2021). http://lse.sourceforge.net/io/aio.html. Last access: May 17, 2023.

[133] Kivity, A. (2017). Different i/o access methods for linux, what we chose for scylla, and why. https://www.scylladb.com/2017/10/05/io-access-methods-scylla/. Last access: May 17, 2023.

[134] Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.".

[135] Klimovic, A., Kozyrakis, C., Thereska, E., John, B., and Kumar, S. (2016). Flash storage disaggregation. In *EuroSys*, pages 1–15.

[136] Klimovic, A., Litz, H., and Kozyrakis, C. (2018). Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*, pages 759–773.

[137] Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A. L., Costa, P., and Pietzuch, P. (2016). Saber: Window-based hybrid stream processing for heterogeneous architectures. In *ACM SIGMOD*, page 555–569.

[138] Kovacs, G. (2017). Ebs, efs, or amazon s3: which is the best cloud storage system for you? https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system. Last access: May 17, 2023.

[139] Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *NetDB*, pages 1–7.

[140] Krishnamurthy, S., Wu, C., and Franklin, M. (2006). On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634.

[141] Lamport, L. (2019). Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196.

[142] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75.

[143] Leis, V., Boncz, P., Kemper, A., and Neumann, T. (2014). Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754.

[144] Leis, V., Kundhikanjana, K., Kemper, A., and Neumann, T. (2015). Efficient processing of window functions in analytical sql queries. *Proc. VLDB Endow.*, 8(10):1058–1069.

[145] Lemire, D. and Boytsov, L. (2015). Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, pages 1–29.

[146] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005a). No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34:39–44.

[147] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005b). Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, page 311–322.

[148] Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., and Maier, D. (2008). Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288.

[149] Lin, W., Qian, Z., Xu, J., Yang, S., Zhou, J., and Zhou, L. (2016). Streamscope: Continuous reliable distributed processing of big data streams. In *USENIX NSDI*, pages 439–453.

[150] Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60.

[151] Marcu, O.-C., Costan, A., Antoniu, G., Pérez-Hernández, M., Nicolae, B., Tudoran, R., and Bortoli, S. (2018). Kera: Scalable data ingestion for stream processing. In *IEEE ICDCS*, pages 1480–1485.

[152] Materialize, t. . R. (2020). https://materialize.io/blog-roadmap/. Last access: May 17, 2023.

[153] Mattern, F. et al. (1988). *Virtual time and global states of distributed systems*. Univ., Department of Computer Science.

[154] Mayer-Schönberger, V. and Cukier, K. (2013). *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt.

[155] McSherry, F., Isard, M., and Murray, D. G. (2015). Scalability! but at what {COST}? In *HotOS*.

[156] Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Pavlo, A., Stonebraker, M., Tufte, K., and Wang, H. (2015). S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13):2134–2145.

[157] Miao, H., Jeon, M., Pekhimenko, G., McKinley, K. S., and Lin, F. X. (2019). Streambox-hbm: Stream analytics on high bandwidth hybrid memory. In *ASPLOS*, pages 167–181.

[158] Miao, H., Park, H., Jeon, M., Pekhimenko, G., McKinley, K. S., and Lin, F. X. (2017). Streambox: Modern stream processing on a multicore machine. In *USENIX ATC*, pages 617–629.

[159] Mitchell, C., Geng, Y., and Li, J. (2013). Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *USENIX ATC)*, pages 103–114.

[160] Mohr-Daurat, H., Theodorakis, G., and Pirk, H. (2022). Boss - an extensible dbms architecture for bare-metal performance. *Under submission in ACM SIGMOD*.

[161] Monte, D., Zeuch, S., Rabl, T., and Markl, V. (2020). Rhino: Efficient management of very large distributed state for stream processing engines. In *ACM SIGMOD*, page 2471–2486.

[162] Mueller, R., Teubner, J., and Alonso, G. (2009). Streams on wires: a query compiler for fpgas. *Proc. VLDB Endow.*, 2(1):229–240.

[163] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: a timely dataflow system. In *ACM SOSP*, pages 439–455.

[164] Muthukrishnan, S. (2005). *Data streams: Algorithms and applications*.

[165] Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. In *Proc. VLDB Endow.*, volume 4, pages 539–550.

[166] OPRA (2021). Opra migration to new opra pillar platform: Capacity testing(revised). https://assets.website-files.com/5ba40927ac854d8c97bc92d7/60bfe785b5efd952686c5dd1_OPRA%202021%20Capacity%20Projections%20and%20Testing%20Dates%20June%2012%202021_Revised.pdf. Last access: May 17, 2023.

[167] Oracle®Stream Explorer (2023). http://bit.ly/1L6tKz3. Last access: May 17, 2023.

[168] Pandis, I., Johnson, R., Hardavellas, N., and Ailamaki, A. (2010). Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939.

[169] Patroumpas, K. and Sellis, T. (2006). Window specification over data streams. In *EDBT*, pages 445–464. Springer.

[170] Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (raid). In *ACM SIGMOD*, pages 109–116.

[171] Pekhimenko, G., Guo, C., Jeon, M., Huang, P., and Zhou, L. (2018). Tersecades: Efficient data compression in stream processing. In *USENIX ATC*, pages 307–320.

[172] Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2012). Base-delta-immediate compression: Practical data compression for on-chip caches. In *ACM PACT*, page 377–388.

[173] Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., and Veeraraghavan, K. (2015). Gorilla: A fast, scalable, in-memory time series database. In *Proc. VLDB Endow.*, volume 8, pages 1816–1827.

[174] Perry, J. S. (2017). What is big data? more than volume, velocity and variety.... https://developer.ibm.com/blogs/what-is-big-data-more-than-volume-velocity-and-variety/. Last access: May 17, 2023.

[175] Pietzuch, P., Garefalakis, P., Koliousis, A., Pirk, H., and Theodorakis, G. (2018). Do we need distributed stream processing? https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. Last access: May 17, 2023.

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

[177] Polychroniou, O., Raghavan, A., and Ross, K. A. (2015). Rethinking simd vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508.

[178] Postgres (2013). Why we are going to have to go directio. https://www.postgresql.org/message-id/529F7D58.1060301%40agliodbs.com. Last access: May 17, 2023.

[179] Protocol Buffers (2022). https://developers.google.com/protocol-buffers/. Last access: May 17, 2023.

[180] Pulsar, A. (2016). Open-sourcing pulsar, pub-sub messaging at scale. https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale. Last access: May 17, 2023.

[181] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. (2013). Timestream: Reliable stream computation in the cloud. In *EuroSys*, pages 1–14.

[182] RabbitMQ (2022). http://rabbitmq.org/. Last access: May 17, 2023.

[183] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. (2017). Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*, pages 497–514.

[184] Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., and et al. (2013). Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091.

[185] Redpanda Raison D'etre (2019). https://vectorized.io/blog/redpanda-raison-detre/. Last access: May 17, 2023.

[186] Reinsel, D., Gantz, J., and Rydning, J. (2018). Data age 2025: The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. Last access: May 17, 2023.

[187] Rödiger, W., Mühlbauer, T., Kemper, A., and Neumann, T. (2015). High-speed query processing over high-speed networks. *arXiv preprint arXiv:1502.07169*.

[188] Rompf, T. and Odersky, M. (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136.

[189] Roth, M. A. and Van Horn, S. J. (1993). Database compression. *ACM Sigmod Record*, pages 31–39.

[190] Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59.

[191] Shaikhha, A., Klonatos, Y., Parreaux, L., Brown, L., Dashti, M., and Koch, C. (2016). How to architect a query compiler. In *SIGMOD*, pages 1907–1922.

[192] Shao, Z. (2011). Real-time analytics at facebook. In *Fifth Conference on Extremely Large Databases, XLDB5*.

[193] Shatdal, A. and Naughton, J. F. (1995). Adaptive parallel aggregation algorithms. In *SIGMOD*, page 104–114.

[194] Shein, A. U., Chrysanthis, P. K., and Labrinidis, A. (2018). Slickdeque: High throughput and low latency incremental sliding-window aggregation. In *EDBT*, pages 397–408.

[195] Silvestre, P. F., Fragkoulis, M., Spinellis, D., and Katsifodimos, A. (2021). Clonos: Consistent causal recovery for highly-available streaming dataflows. In *ACM SIGMOD*, page 1637–1650.

[196] Stonebraker, M. (1986). The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9.

[197] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM Sigmod Record*, pages 42–47.

[198] Strom, R. and Yemini, S. (1985). Optimistic recovery in distributed systems. *ACM TOCS*, pages 204–226.

[199] Syinchwun, L. (2016). Lightweight event time window. in flink jira issues. https://issues.apache.org/jira/browse/FLINK-5387. Last access: May 17, 2023.

[200] Tangwongsan, K., Hirzel, M., and Schneider, S. (2017). Low-latency sliding-window aggregation in worst-case constant time. In *DEBS*, page 66–77.

[201] Tangwongsan, K., Hirzel, M., and Schneider, S. (2019). Optimal and general out-of-order sliding-window aggregation. *Proc. VLDB Endow.*, 12(10):1167–1180.

[202] Tangwongsan, K., Hirzel, M., Schneider, S., and Wu, K.-L. (2015). General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713.

[203] Terry, D., Goldberg, D., Nichols, D., and Oki, B. (1992). Continuous queries over append-only databases. *Sigmod Record*, 21(2):321–330.

[204] Theodorakis, G., Koliousis, A., Pietzuch, P., and Pirk, H. (2018). Hammer slide: work-and cpu-efficient streaming window aggregation. pages 34–41.

[205] Theodorakis, G., Koliousis, A., Pietzuch, P. R., and Pirk, H. (2020a). LightSaber: Efficient Window Aggregation on Multi-core Processors. In *ACM SIGMOD*, page 2505–2521.

[206] Theodorakis, G., Kounelis, F., Pietzuch, P. R., and Pirk, H. (2022). Scabbard: Single-Node Fault-Tolerant Stream Processing. VLDB '22. ACM.

[207] Theodorakis, G., Pietzuch, P. R., and Pirk, H. (2020b). Slideside: A fast incremental stream processing algorithm for multiple queries. In *EDBT*, pages 435–438.

[208] Thomas, J., Hanrahan, P., and Zaharia, M. (2020). Fleet: A framework for massively parallel streaming on fpgas. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 639–651.

[209] Ting, D. (2018). Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD*, pages 1129–1140.

[210] To, Q.-C., Soto, J., and Markl, V. (2018). A survey of state management in big data processing systems. *The VLDB Journal*, pages 847–872.

[211] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm@Twitter. In *ACM SIGMOD*, pages 147–156.

[212] Traub, J., Grulich, P. M., Cuellar, A. R., Breß, S., Katsifodimos, A., Rabl, T., and Markl, V. (2018). Scotty: Efficient window aggregation for out-of-order stream processing. In *ICDE*, pages 1300–1303. IEEE.

[213] Trident (2022). http://storm.apache.org/Trident-tutorial.html. Last access: May 17, 2023.

[214] Tucker, P., Tufte, K., Papadimos, V., and Maier, D. (2008). Nexmark—a benchmark for queries over data streams draft. Technical report, Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[215] Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. (2003). Exploiting punctuation semantics in continuous data streams. *TKDE*, 15:555–568.

[216] uv 300, S. (2015). https://www.earlham.ac.uk/sgi-uv300. Last access: May 17, 2023.

[217] Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M. J., Recht, B., and Stoica, I. (2017). Drizzle: Fast and adaptable stream processing at scale. In *ACM SOSP*, page 374–389.

[218] Viglas, S. D. and Naughton, J. F. (2002). Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48.

[219] Wang, S., Liagouris, J., Nishihara, R., Moritz, P., Misra, U., Tumanov, A., and Stoica, I. (2019). Lineage stash: fault tolerance off the critical path. In *ACM SOSP*, pages 338–352.

[220] Wilkes, J. (2011). More Google Cluster Data. Google Research Blog. http://bit.ly/1A38mfR. Last access: May 17, 2023.

[221] Williams, R. N. (1991). An extremely fast ziv-lempel data compression algorithm. In *IEEE Data Compression Conference*, pages 362–363.

[222] Wu, Y. and Tan, K.-L. (2015). Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734.

[223] Xie, D., Chandramouli, B., Li, Y., and Kossmann, D. (2019). Fishstore: Faster ingestion with subset hashing. In *ACM SIGMOD*, pages 1711–1728.

[224] Xing, Y., Zdonik, S., and Hwang, J.-H. (2005). Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802.

[225] Xu, L., Venkataraman, S., Gupta, I., Mai, L., and Potharaju, R. (2021). Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *NSDI*, pages 389–405.

[226] Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V. (2015). Performance analysis of nvme ssds and their implication on real world databases. In *ACM International Systems and Storage Conference*, pages 1–11.

[227] Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., and Ganguli, D. (2014). Druid: A real-time analytical data store. In *SIGMOD*, pages 157–168.

[228] Yavuz, B. (2017). Benchmarking structured streaming on databricks runtime against state-of-the-art streaming systems. https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html. Last access: May 17, 2023.

[229] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278.

[230] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, pages 15–28.

[231] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *ACM SOSP*, pages 423–438.

[232] Zeitler, E. and Risch, T. (2011). Massive scale-out of expensive continuous queries. *Proc. VLDB Endow.*, 4(11):1181–1188.

[233] zeromq (2022). http://zeromq.org/. Last access: May 17, 2023.

[234] Zeuch, S., Monte, B. D., Karimov, J., Lutz, C., Renz, M., Traub, J., Breß, S., Rabl, T., and Markl, V. (2019). Analyzing efficient stream processing on modern hardware. In *Proc. VLDB Endow.*, volume 12, pages 516–530.

[235] Zhang, S. (2019). *Scaling data stream processing on multicore architectures*. PhD thesis, School of Computing National University of Singapore.

[236] Zhang, S., He, J., Zhou, A. C., and He, B. (2019). Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *ACM SIGMOD*, pages 705–722.

[237] Zhang, S., Mao, Y., He, J., Grulich, P. M., Zeuch, S., He, B., Ma, R. T. B., and Markl, V. (2021). *Parallelizing Intra-Window Join on Multicores: An Experimental Study*, page 2089–2101.

[238] Zhang, S., Zhang, F., Wu, Y., He, B., and Johns, P. (2020). Hardware-conscious stream processing: A survey. *SIGMOD Record*, 48(4):18–29.

[239] Zhang, Y. and Mueller, F. (2011). Gstream: A general-purpose data streaming framework on gpu clusters. In *ICPP*, pages 245–254.

[240] Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. (2004). Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, pages 431–442.

[241] Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, pages 337–343.

Appendix A

Workloads

A.1 Benchmark queries

A.1.1 Cluster monitoring

```
-- Query 1
--
-- Input:  TaskEvents
--         long  timestamp
--         long  jobId
--         long  taskId
--         long  machineId
--         int   eventType
--         int   userId
--         int   category
--         int   priority
--         float cpu
--         float ram
--         float disk
--         int   constraints
-- Output: CPUusagePerCategory
--         long  timestamp
--         int   category
--         float totalCpu
--
select timestamp, category, sum(cpu) as totalCpu
from TaskEvents [range 60 slide 1]
group by category
--
-- Query 2
--
-- Input:  TaskEvents
-- Output: CPUusagePerJob
--         long  timestamp
--         long  jobId
--         float avgCpu
--
```

```
select timestamp, jobId, avg(cpu) as avgCpu
from TaskEvents [range 60 slide 1]
where eventType == 1
group by jobId
```

A.1.2 Smart grid

```
-- Query 1
--
-- Input:  SmartGridStr
--         long  timestamp
--         float value
--         int   property
--         int   plug
--         int   household
--         int   house
-- Output: GlobalLoadStr
--         long  timestamp
--         float globalAvgLoad
--
select timestamp,  avg(value) as globalAvgLoad
from SmartGridStr [range 3600 slide 1]
--
-- Query 2
--
-- Input:  SmartGridStr
-- Output: LocalLoadStr
--         long  timestamp
--         int   plug,
--         int   household
--         int   house
--         float localAvgLoad
--
select timestamp, plug, household, house, avg(value) as localAvgLoad
from SmartGridStr [range 128 slide 1]
group by plug, household, house
--
-- Query 3
--
-- Input:  GlobalLoadStr, LocalLoadStr
-- Output: Outliers
--         long  timestamp
--         int   house
--         float count
--
(
select L.timestamp, L.plug, L.household, L.house
from LocalLoadStr  [range 1 slide 1] as L,
     GlobalLoadStr [range 1 slide 1] as G
where L.localAvgLoad > G.globalAvgLoad
) as R
--
select timestamp, house, count(*)
```

```
from R
group by house
```

A.1.3 Linear Road Benchmark

```
-- Query 0
--
-- Input:  PosSpeedStr
--         long  timestamp
--         int   vehicle
--         float speed
--         int   highway
--         int   lane
--         int   direction
--         int   position
-- Output: SegSpeedStr
--         long  timestamp
--         int   vehicle
--         float speed
--         int   highway
--         int   lane
--         int   direction
--         int   segment
--
select timestamp, vehicle, speed, highway, lane, direction, (position/5280) as
    segment
from PosSpeedStr [range unbounded]
--
-- Query 1
--
-- Input:  SegSpeedStr
-- Output: CongestedSegRel
--         long  timestamp
--         int   highway
--         int   direction
--         int   segment
--         float avgSpeed
--
select timestamp, highway, direction, segment, avg(speed) as avgSpeed
from SegSpeedStr [range 300 slide 1]
group by highway, direction, segment
having avgSpeed < 40.0
--
-- Query 2
--
-- Input:  SegSpeedStr
-- Output: SegVolRel
--         long  timestamp
--         int   highway
--         int   direction
--         int   segment
--         float numVehicles
--
```

```
select timestamp, vehicle, highway, direction, segment, count(*)
from SegSpeedStr [range 30 slide 1]
group by highway, direction, segment, vehicle
--
-- Query 3
--
-- Input:  SegSpeedStr
-- Output: SegVolRel
--         long  timestamp
--         int   highway
--         int   direction
--         int   segment
--         float numVehicles
--
(
  select timestamp, vehicle, highway, direction, segment, count(*)
  from SegSpeedStr [range 30 slide 1]
  group by highway, direction, segment, vehicle
) as R
--
select timestamp, highway, direction, segment, count(vehicle) as numVehicles
from R
group by highway, direction, segment
```

A.1.4 Yahoo Streaming Benchmark

```
-- Query 1
--
-- Input:  InputStr
--         long  eventTime
--         long  userId
--         long  pageId
--         long  adId
--         long  adType
--         long  eventType
--         long  ipAddress
-- Output: OutputStr
--         int   numCampaigns
--         long  lastUpdate
--
select count (*), max(eventTime) as lastUpdate
from
(
     select campaignId, RS.adId, eventTime
     from InputStream [range 10 slide 10] as IS
     join Campaigns as C
     on C.adId = IS.adId
     where eventType = "view"
)
group by campaignId
```

A.1.5 Sensor Monitoring

```
-- Query 1
--
-- Input:  SensorStr
--         long  timestamp
--         int   messageIndex
--         int   mf01
--         int   mf02
--         int   mf03
--         int   pc13
--         int   pc14
--         int   pc15
--         unsigned int pc25
--         unsigned int pc26
--         unsigned int pc27
--         unsigned int res
--         bool bm05
--         bool bm06
--         bool bm07
--         bool bm08
--         bool bm09
--         bool bm10
-- Output: OutputStr
--         long  timestamp
--         int   gloabalAvgMf01
--         int   gloabalAvgMf02
--         int   gloabalAvgMf03
--
select timestamp, avg(mf01), avg(mf02), avg(mf03)
from SensorStr [range 60 slide 1]
--
-- Query f
--
-- Input:  SensorStr
-- Output: OutputStr
--         long  timestamp
--         int   aggregateFunction
--
select timestamp, f(mf01) as aggregateFunction
from SensorStr [range (_) slide 1]
```

A.1.6 NEXMark

```
-- Query 1
--
-- Input:  BidStr
--         long timestamp;
--         long id;
--         long itemName;
--         long description;
--         long initialBid;
--         long reserve;
```

```
--          long expires;
--          long seller;
--          long category;
-- Output: OutputStr
--          long timestamp
--          long id
--
select id
from bid [range 60 slide 1]
where
(
  select count(id)
  from bid [range 60 slide 1]
  group by id
) >=
all (
  select count(id)
  from bid [range 60 slide 1]
  group by id
)
```