Hardware-conscious Techniques for Efficient and Reliable Stateful Stream Processing

vorgelegt von
M. Sc.
Bonaventura Del Monte

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Stefan Schmid
Gutachter: Prof. Dr. Volker Markl
Gutachter: Prof. Peter Pietzuch, Ph.D.
Gutachter: Prof. Matei Zaharia, Ph.D.
Gutachter: Prof. Dr. Tilmann Rabl
Tag der wissenschaftlichen Aussprache: 5. Dezember 2022

Berlin 2023

Zusammenfassung

In den letzten zwei Jahrzehnten wurden verteilte Datenflussverarbeitungssysteme zu einer wichtigen Komponente in dem Big-Data-Verwaltungs-Toolkit, um zustandsabhängige Echtzeit-Datenanalyseanwendungen für hochvolumige Datenströme mit hoher Geschwindigkeit in Cloud-Bereitstellungen zu unterstützen. Zu diesem Zweck führen aktuelle Datenflussverarbeitungssysteme kontinuierlich Map/Reduce-ähnliche Pipelines auf kontinuierlichen Daten aus und wenden datenzentrische Parallelität zur Skalierung auf einem Server-Cluster an. Aktuelle Datenflussverarbeitungssysteme setzen sogenannte Commodity-Hardware voraus, da sie Map/Reduce-ähnlichen Paradigmen folgen, die auf der Shared-Nothing-Architektur basieren. Darüber hinaus sind Datenflussverarbeitungssysteme unabhängig von der Hardwarekonfiguration, da sie sich auf verwaltete Laufzeiten, wie z. B. eine Java Virtual Machine, stützen. Moderne Hardware-Infrastrukturen haben sich jedoch in den letzten Jahren dramatisch verbessert. Daher ist die gängige Meinung, dass Cloud-Anbieter hauptsächlich Standard-Hardware anbieten, nicht mehr gültig. So bieten die Anbieter von Cloud-Plattformen leistungsstarke Rechen- und Netzwerkfunktionen, da sie Server mit High-End-CPUs mit vielen Kernen und großen Caches sowie Hochgeschwindigkeitsnetzwerke wie Infiniband mit RDMA-Unterstützung (Remote Direct Memory Access) anbieten. Darüber hinaus sind moderne verteilte Recheninfrastrukturen äußerst flexibel, da sie eine Ad-hoc-Bereitstellung von Ressourcen ermöglichen, die eine Skalierung der Rechen- und Speicherkapazitäten sowie die Behandlung von Ausfällen ermöglichen, während eine Anwendung ausgeführt wird.

In dieser Arbeit zeigen wir, dass die aktuelle Generation von Datenflussverarbeitungssysteme hardware-agnostisch ist und die oben genannten technologischen Fortschritte nicht nutzen kann. Wir zeigen experimentell, dass diese ineffizient arbeiten, wenn sie auf einer Infrastruktur ausgeführt werden, die CPUs auf dem Leistungsniveau für HPC-Anwendungen, Hochgeschwindigkeitsnetzwerkkommunikation sowie eine flexible Ad-hoc-Ressourcenbereitstellung bietet. Zu diesem Zweck stellen wir Lösungen zur effizienten Ausführung von zustandsabhängigen Stream-Processing-Anwendungen auf einer modernen Recheninfrastrukturen vor.

Zunächst konzentrieren wir uns auf die Skalierungsleistung aktueller Datenflussverarbeitungssysteme, um die Rechenkapazitäten der modernen Hardware zu nutzen. Unsere Analyse zeigt, dass Datenflussverarbeitungssysteme unter ineffizienten Speicherzugriffsmustern leiden, die zu einer suboptimalen Code- und Datenlokalität führen. Auf der Grundlage unserer Analyse schalgen wir Designänderungen, wie z. B. eine spezialisierte Codegenerierung, vor um die allgemeinen

Architektur dahingehend zu ändern das ein SPE eine Skalierung auf moderner Hardware ermöglicht.

Zweitens konzentrieren wir uns auf die Scale-out-Leistung der aktuellen Datenflussverarbeitungssysteme, um die Hochgeschwindigkeitsnetzwerke mit RDMA-Unterstützung zu nutzen. Insbesondere die RDMA-Hardware hat die gängige Annahme entkräftet, dass das Netzwerk in verteilten Datenverarbeitungssystemen oft ein Engpass ist. Hochgeschwindigkeitsnetze bieten jedoch keine Plug-and-PlayLeistung (z. B. bei der Verwendung von IP-over-InfiniBand) und erfordern eine sorgfältige gemeinsame Entwicklung von System- und Anwendungslogik. Insgesamt erreicht unsere Lösung eine Durchsatzverbesserung um bis zu zwei Größenordnungen gegenüber bestehenden Systemen, die in einem InfiniBand-Netzwerk eingesetzt werden, und sie ist bis zu einem Faktor von 22 schneller als eine selbst entwickelte Lösung, die auf RDMA-basierter Datenvorpartitionierung zur Skalierung der Abfrageverarbeitung beruht.

Abschließend konzentrieren wir uns auf die Laufzeit-Rekonfiguration laufender Streaming-Anwendungen der aktuellen Generation von Datenstromverarbeitungssystemen, um die Ad-hoc- und flexiblen Bereitstellungsmöglichkeiten der modernen Cloud-Computing-Infrastruktur zu nutzen. Datenflussverarbeitungssysteme müssen zustandsbehaftete Abfragen während der Laufzeit rekonfigurieren, um sich von Ausfällen zu erholen, sich an schwankende Datenraten anzupassen und eine Verarbeitung mit geringer Latenz zu gewährleisten, was in industriellen Umgebungen erforderlich ist. Modernste Datenflussverarbeitungssysteme sind jedoch noch nicht in der Lage, Abfragen mit Terabytes an Zuständen während der Laufzeit zu rekonfigurieren, was auf drei Probleme zurückzuführen ist: Netzwerk-Overhead für Zustandsmigration, Konsistenz und Mehrkosten bei der Datenverarbeitung. Wir schlagen *Rhino* vor, ein System für die effiziente Rekonfiguration laufender Abfragen in Anwesenheit eines verteilten Zustands beliebiger Größe. Insgesamt zeigt unsere Untersuchung, dass Rhino mit Zustandsgrößen von bis zu mehreren Terabytes skaliert, eine laufende Abfrage 15-mal schneller rekonfiguriert als die modernsten Lösungen und die Latenz bei einer Rekonfiguration um drei Größenordnungen reduziert.

Zusammenfassend lässt sich sagen, dass diese Arbeit die Grundlage für eine effiziente und zuverlässige zustandsbehaftete Stream-Verarbeitung durch ein hardwarebewusstes Systemdesign legt, das auf bestehende und zukünftige SPEs angewendet werden kann. Durch unser neuartiges Systemdesign erreichen unsere Software-Prototypen, die in dieser Arbeit vorgeschlagen wurden, eine überlegene Leistung im Vergleich zu modernen Datenstromverarbeitungssystemen bei gängigen Datenstromverarbeitungsaufgaben.

Abstract

Over the past two decades, distributed stream processing engines (SPEs) have become a prominent component in the big data management tool-chain to support real-time, stateful data analytics applications on high-volume, high-velocity data streams in cloud deployments. To this end, current SPEs continuously execute Map/Reduce-like pipelines on continuous data and apply data-centric parallelism to scale-out on a cluster of servers. Current SPEs assume so-called commodity hardware as they follow Map/Reduce-like paradigms based on the shared-nothing architecture. Furthermore, SPEs are agnostic to hardware configuration as they rely on managed runtimes, such as a Java Virtual Machine. However, computing infrastructures have improved dramatically their hardware characteristics in the past years. As a result, the common wisdom that cloud providers mainly offer commodity hardware no longer holds. For instance, cloud platform vendors provide powerful compute and network capabilities as they offer servers with high-end CPUs with many cores and large caches as well as high-speed networks, such as Infiniband with Remote Direct Memory Access (RDMA) support. Furthermore, modern hardware infrastructure is highly flexible, as it provides ad-hoc provisioning of resources, which enables scaling the compute and storage capabilities as well as coping with failures, while a deployed application is executed.

In this thesis, we show that the current generation of SPEs are hardware-agnostic and cannot leverage the above technology advancements. In fact, we experimentally demonstrate that they perform inefficiently when running on an infrastructure that provides HPC-grade CPUs, high-speed networks, as well as ad-hoc, flexible resource provisioning. To this end, we present solutions to efficiently execute stateful stream processing applications on the modern hardware infrastructure. First, we focus on the scale-up performance of current SPEs to leverage the compute capabilities of the modern hardware. Our analysis shows that SPEs suffer from inefficient memory access patterns that lead to sub-optimal code and data locality. Driven by our analysis, we provide design changes, such as specialized code generation, to the common architecture of an SPE to scale-up on modern hardware. We show that an SPE that follows our guidelines achieves up to two orders of magnitude higher single-node throughput compared to state-of-the-art SPEs.

Second, we focus on the scale-out performance of the current SPEs to leverage the high-speed networks with RDMA support. In particular, RDMA hardware has invalidated the common assumption that network is often a bottleneck in distributed data processing systems. However, high-speed networks do not provide "plug-and-play" performance (e.g., using IP-over-InfiniBand) and require a careful co-design of system and application logic. To this end, we propose *Slash*, a

novel stream processing engine that uses high-speed networks and RDMA to efficiently execute distributed streaming computations. Overall, our solution achieves a throughput improvement up to two orders of magnitude over existing systems deployed on an InfiniBand network and it is up to a factor of 22 faster than a self-developed solution that relies on RDMA-based data pre-partitioning to scale out query processing.

Finally, we focus on the runtime reconfiguration of running streaming applications of the current generation of SPEs to leverage the ad-hoc and flexible provisioning capabilities of the modern cloud computing infrastructure. SPEs need to transparently reconfigure stateful queries during runtime to recover from outages, adjust to varying data rates, and ensure low-latency processing, which are required by industrial setups. However, state-of-the-art SPEs are not ready yet to handle on-the-fly reconfiguration of queries with terabytes of state due to three problems: network overhead for state migration, consistency, and overhead on data processing. We propose *Rhino*, a library for efficient reconfiguration of running queries in the presence of distributed state of arbitrary size. Overall, our evaluation shows that Rhino scales with state sizes of up to terabytes, reconfigures a running query 15 times faster than the state-of-the-art solutions, and reduces latency by three orders of magnitude upon a reconfiguration.

In sum, the thesis lays the foundation for efficient and reliable stateful stream processing via a hardware-conscious system design that can be applied to existing and future SPEs. Through our novel system design, the software prototypes proposed in this work achieve superior performance compared to state-of-the-art SPEs on common stream processing workloads.

Acknowledgements

I will be forever grateful to everyone who supported and nurtured my academic and professional growth. First and foremost, I am deeply indebted to my advisors Volker Markl and Tilmann Rabl, who took me under their wings as doctoral student and guided me towards success. In the last phase of my studies, Peter Pietzuch, Matei Zaharia, as well as Stefan Schmid, honored me greatly by forming my doctoral committee along with Volker Markl and Tilmann Rabl. I am very grateful for their counsel, as it helped shape this thesis as well as fostered new ideas for my future career path.

Throughout this time, Tilmann Rabl first and later with the help of Steffen Zeuch and Sebastian Breß mentored me on high-impact research. They helped me find my research direction and reviewed my early paper drafts. I could not have completed this thesis without their support.

During my time as a Ph.D. student, people at the Database and Information Systems Group of TU Berlin accompanied me and worked with me to move forward my as well the team research agenda. I especially thank Clemens Lutz, Alireza Rezaei Mahdiraji, Dimitrios Giouroukis, Haralampos Gavriilidis, Jeyhun Karimov, Behrouz Derakhshan, Philipp Grulich, Martin Kiefer, Gabor Gevay, Alexander Renz-Wieland, Andreas Kunft, and Viktor Rosenfeld, with whom I reflected on research ideas.

Furthermore, I thank the former and present members of the NebulaStream team at TU Berlin for the great collaboration on research and engineering topics related to NebulaStream: Ankit Chaudhary, Ariane Ziehn, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Dwi Prasetyo Adi Nugroho, Varun Pandey, Nils Schubert, Maroua Taghouti, Julius Hülsmann, Anastasiia Kozar, Adrian Michalke, Aljoscha Lepping, Jonas Traub as well as Dimitrios Giouroukis, Haralampos Gavriilidis, Philipp Grulich, Viktor Rosenfeld, and Steffen Zeuch.

Additionally, I would like to thank past and present senior researchers in the team for their feedback on this thesis, help with my Ph.D. talk, and their career advices: Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, Kaustubh Beedkar, Abdulrahman Kaitoua, Juan Soto, Asterios Katsifodimos, and Ziawasch Abedjan.

I also wish to thank the colleagues at DIMA, Serafeim Papadias, Kajetan Maliszewski, Sergey Redyuk, Rudi Poepsel Lemaitre, Mahdi Esmailoghli, Mohammad Mahdavi, Felix Neutatz, Ricardo Ernesto Martinez Ramirez, and Lennart Behme, for the interesting discussions we had over lunch or in front of a coffee at TU Berlin.

I also thank the admin staff at TU Berlin, Claudia Gantzer and Melanie Neumann, who helped me dealing with the ”complex” paperwork of TU Berlin as well as Lutz Friedel for his help with the DIMA server fleet.

I would also like to thank all my co-authors: Steffen Zeuch, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl, Adrian Bartnik, Hendrik Makait, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Dimitrios Giouroukis, Philipp Grulich, Ariane Ziehn, and Haralampos Gavriilidis.

Last but not least, I would like to thank my friends who aided and encouraged me during the Ph.D. time. In particular, I wish to thank my friends Pietro C., Pietro S., Piero, Francesco, Antonio, Jasmin, Chiara, Maria, Dino, Gianmarco, Michele, Tiziana, Luca, Emanuela, Vivien, Alireza, Behrouz, Valeria, Adriano, Amedeo, Marco, Giulio, Daniele, Gianluca, Gabriele, Giuseppe, Sofia, Maurizio, and Lino as well as the members of the ”South Cartel” club: Dimitri, Makis, and Hari. We spent lots of time discussing and arguing about every little thing on earth in front of good Italian and Greek food, beers, and glasses of wine.

Finally, I happened to get closer to Silvia in the last years of my PhD and she gave me a lot of support and careless moments. She has a lot of merit in this thesis, as she has been my source of strength and confidence.

I dedicate this thesis to my family, to my parents Giuseppina and Carlo, to my uncle and aunt Franco and Marina, to my cousins Marco and Christian, and to my grandparents Gerardo and Cristina. They defined the way I see the world today and gave me trust, love, care and strong principles. I wish my father lived long enough to see this thesis completed. Thank you all for everything you have done for me.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Over the past decades, stateful stream processing has become an important data processing paradigm in the big data management tool-chain, as it enables real-time and scalable analysis over high-volume, high-speed streams of continuous data. To meet the requirements of data-intensive applications, both academia and industry have proposed several Stream Processing Engines (SPEs) as general-purpose frameworks to express and execute distributed, stateful computations over unbounded data streams. Stateful computations involve persistent state that an SPE continuously accesses, while it processes continuous data. Research distinguishes two classes of SPEs: scale-out and scale-up SPEs. Scale-out SPEs target scalability and execute queries over a cluster of nodes, whereas scale-up SPEs target maximum performance on a single-node. Prominent scale-out SPEs are Apache Flink [1], Apache Spark [2], and Apache Heron [3]. Scale-up SPEs are Streambox [4], Lightsaber [5], and Briskstream [6].

Nowadays, SPEs power many stateful analytical applications behind popular multimedia services, online marketplaces, cloud providers, and mobile games. For instance, these services deploy SPEs to implement a wide range of data-driven use-cases, e.g., fraud detection, content recommendation, and user profiling [7, 8, 9, 1, 10, 11, 12]. Online marketplaces and games monitor their users to prevent frauds and offer ad-hoc purchases [7, 1]. Media services dynamically recommend new content while analyzing user activities [9], which result into up to one trillion stream records/day [13]. Media services analyze the behavior of their users to provide insights as well as recommend new content and in-app purchases [8, 9, 10, 11, 12]. Furthermore, cloud providers offer fully-managed SPEs to customers to implement their use cases, which hide operational details [14, 15].

To run data-driven applications, SPEs have to support continuous stateful stream processing under diverse conditions, such as fluctuating data rates and tight latency requirements. Thus,

SPEs need to provide robust and scalable query processing performance, while guaranteeing reliable and consistent stateful computations. However, this is a multifaceted, open problem in stateful stream processing that current SPEs still fall short to address, as we will show in this thesis. Current SPEs are *hardware-oblivious* and do not consider the intricacies of modern hardware resources, such as multi-core CPUs with large caches and high-speed networks. As a consequence, they require large clusters to scale out stateful stream processing on real-world data streams, as they provide poor single-node performance [16]. Furthermore, SPEs are not ready yet to transparently react to anomalous operational events and adjust their processing capabilities, accordingly. As a result, they require a restart of the affected queries. This results in downtime and thus in high latency that hinders analytics readiness and induces high infrastructure cost.

Motivated by the above challenges and industrial requirements, the goal of this thesis is to provide hardware-conscious techniques to enable efficient and reliable stateful stream processing. By benchmarking the current generation of SPEs on common workloads, we identify three research problems that fundamentally prevent them to achieve robust query processing performance. We list the three problems in the following. First, SPEs trade scale-up single-node performance for scalability to large clusters and use managed runtimes for platform independence, such as the Java Virtual Machine (JVM), which hinders full CPU exploitation. Second, SPEs are designed around a shared-nothing architecture that does not scale on high-speed networks due to computational bottlenecks, even if the executed workloads are data-intensive. Finally, they do not provide efficient on-the-fly reconfiguration of running queries and large state migration techniques to enable timely reactions to anomalous events, such as failures and fluctuations in the data rate of the input data streams. In sum, these problems cause severe performance issues, as we will show throughout this thesis.

Our solution consists of *architectural changes to the core runtime components of an SPE to enable robust query processing performance on data-intensive workloads.* In particular, we devise a *hardware-conscious* system design that provides building blocks for efficient *scale-up* and *scale-out* processing to fully leverage modern CPU architectures and high-speed networks. Furthermore, we propose a set of protocols to enable state migration and on-the-fly query reconfigurations, which do not halt query execution and guarantee exactly-once processing semantics. Overall, the result of this thesis is a hardware-conscious SPE architecture that improves the query processing performance of state-of-the-art stream processing technology.

1.2 Research Problems and Contributions

In this section, we state the research problems addressed in this thesis and outline our contributions. The following chapters discuss our contributions in detail.

In Figure 1.1, we show the common software infrastructure of an SPE, which comprises multiple components, such as the query processing and query optimizer components. In this thesis, we focus on the query processing component, which we divide in three runtimes: the query execution runtime, the distributed dataflow runtime, and the state management runtime.

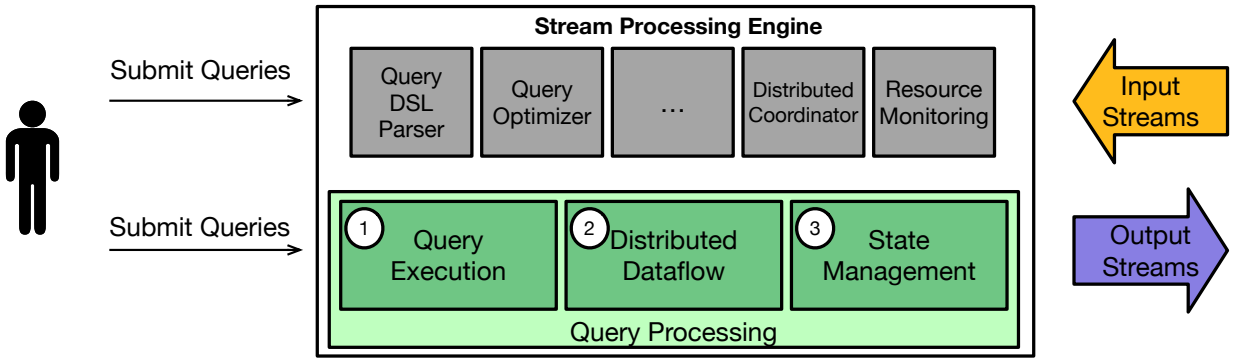


Figure 1.1: High-level software infrastructure of an SPE. Components marked in green are the target of investigation of this thesis.

For our following considerations, we assume that SPEs execute logical query plans that are translated into processing pipelines consisting of source operators, intermediate operators, and sink operators. Source operators continuously provide input data to the processing pipeline, whereas sink operators write out the result stream. Intermediate operators apply processing and exchange data following a producer-consumer pattern: upstream operators produce records that downstream operators consume. The SPE runs each pipeline in parallel over one (scale-up processing) or multiple machines (scale-out processing).

The central runtime component is the query execution runtime ①, which applies stateful query logic on the records of a data stream on a single node. The query execution runtime executes operator logic as a set of data-centric transformations following either a micro-batching or a tuple-at-a-time processing model [17]. The distributed dataflow runtime ② enables scaling out stateful streaming computations over a set of interconnected nodes. Following the dataflow model [15, 18, 19], the SPE executes each operator on multiple nodes and performs data exchange among running operators. The state management runtime ③ controls the life-cycle of a stateful computation. It ensures reliable query execution and comprises solutions for the storage of the state as well as fault-tolerance, resource elasticity, and online query re-optimization.

Overall, the above three runtimes need to provide efficiency, reliability, as well as robust performance to meet the tight requirements of stream processing. However, we show in this thesis that the above runtimes suffer from bottlenecks, which induce performance issues and hinder the robustness of SPEs. In the following, we discuss each bottleneck in detail and identify the underlying research problems that prevent current SPEs from achieving the desired level of performance and robustness. Furthermore, we present our solutions to the identified research problems that system builders may adopt in their SPEs.

1.2.1 Query Execution Runtime

Modern scale-out SPEs, such as Apache Flink [20], Apache Spark [2], Timely Dataflow [21], and Apache Storm [22], distribute processing over a large number of computing nodes in a cluster, i.e., they *scale out* the processing. To process a high volume of data at high speed, their design

goal is to trade single node performance for scalability to large clusters. To this end, they rely on an interpretation-based execution strategy as well as message passing and use managed runtimes as the underlying processing environment for platform independence.

In this thesis, we show that the above approach suffer from inefficient memory access patterns and query execution, which result in low scale-up efficiency. In particular, we analyze the performance robustness of representative SPEs and identify the following two problems: inefficient query execution and inefficient memory access patterns.

Problem 1: Inefficient Query Execution

Current SPEs rely on *interpretation-based execution* to execute queries. This involves an *interpretation-based evaluation* of the query plan, the *application of queues* for passing data as messages, and the application of *operator fusion* based on function calls.

We show that interpretation-based query execution is a source of inefficiency for SPE, as it induces low instruction- and data locality. Interpretation-based query execution relies on two building blocks to scale out: the producer-consumer pattern and message queues. Queues are necessary to implement data exchange among producer and consumer pipelines following a partitioning technique, such as hash-partitioning. However, this approach suffers from a bottleneck, when the producer is faster than the consumer or the distribution of the partitioning keys of the input data is skewed. This performance issue is exacerbated by future network technologies, as they deliver data at a data-rate closed to main-memory speed [23]. Overall, we show in this thesis that current SPEs cannot utilize the memory bandwidth and computational power of today's hardware and, thus, cannot exploit upcoming fast network technologies.

Problem 2: Inefficient Access Patterns to Memory

SPEs that are designed to run on a managed runtime suffer from data-dependent memory accesses. As a result, executed queries suffer from low instruction- and data locality, regardless of the query execution approach.

The query execution runtime of the majority of current SPEs is designed around an execution model based on a common managed-runtime, i.e., the Java Managed Runtime (JVM). In this thesis, we show that SPEs developed using a JVM-based programming language, such as Java, cause a larger number of random memory accesses and virtual function calls compared to C++ implementations. The reason behind this inefficiency is the overhead induced by data (de-)serialization, pointer chasing in data structures, objects scattering in main memory, virtual functions, locking, and garbage collection. Inefficient memory access patterns render the scale-up performance of the query execution runtime of modern SPEs to be severely limited in throughput and latency. Consequently, a scale-out SPE requires a large number of nodes in a cluster to process a query efficiently.

Proposed solution. To address the above problems, we explore design alternatives in-depth and describe a set of design changes that can be incorporated into current and future SPEs. In particular, we propose compilation-based query execution for stream processing workloads,

omit queues, and use late merging instead of partitioning. Furthermore, we propose lock-free windowing instead of lock-based windowing. While the application of compilation-based query execution to relational databases has been deeply investigated over the past decade [24], we are the first to explore the opportunities and challenges behind query compilation for stream processing. Overall, our evaluation shows that scale-up is a viable way of achieving high throughput and low latency stream processing. Our experiments reveal that a scale-up SPE that follows our techniques outperforms a 10-node cluster running state-of-the-art scale-out SPEs and pave the way towards a new generation of scale-up SPEs.

1.2.2 Distributed Dataflow Runtime

The distributed dataflow runtime enables the SPE to scale out query processing over a cluster of nodes. Its key challenge is to ensure efficient distributed query execution over a network of nodes. To this end, it enables data-parallel query execution via *Operator Fission* [25], i.e., the SPE re-partitions the records of data stream according to a partitioning key. Furthermore, the SPE provides each running pipeline with data channels that abstract the network connections among the nodes as well as the in-memory queues. Data channels enable producer pipelines to send a batch of stream records to consumer pipelines based on an exchange strategy, such as hash-partitioning. As a result, consumer pipelines process disjoint partitions of a data stream and may thus compute local states, efficiently.

In this thesis, we show that the distributed dataflow runtime of current SPEs cannot fully utilize the upcoming network technologies that are able to deliver data at memory-speed [23]. One key technology that enables data transfer at high speeds are InfiniBand (IB) or Converged Ethernet networks with Remote Direct Memory Access (RDMA) support. The trend in the area of networking technologies shows that RDMA-capable networks have become more affordable and thus are offered by an increasing number of cloud providers and data centers [26].

The design choices behind their distributed dataflow runtime fundamentally prevent the SPEs from processing data at full network speed for two reasons: kernel-space networking and expensive data shuffling.

Problem 3: Kernel-space Networking

Current SPEs are designed around common networks, such as Ethernet, and rely on socket-based networking, such as TCP/IP, to ingest and exchange data streams. Even though socket-based networking runs on high-speed network hardware, it cannot fully exploit its potential, e.g., using RDMA.

Current SPEs rely on data channels that are designed for common Ethernet networks. However, a highly-efficient SPE needs to exploit the high bandwidth of RDMA hardware and thus support data processing at network line rate. In this thesis, we show that current SPEs are not ready yet for RDMA acceleration, as they cannot fully utilize modern network with RDMA support through a "plug-and-play" approach. However, a drop-in replacement of socket-based networking with RDMA acceleration marginally increases the query processing performance of

an SPE, as our evaluation shows. The architecture of today's SPEs is not suitable for the increase in network bandwidth and requires changes to fully benefit from RDMA, as we discuss in the following.

Problem 4: Expensive Data Shuffling

Data Shuffling for data stream partitioning is inherently compute bound, even in the presence of high-speed networks. In fact, data shuffling via message passing induces expensive queue-based synchronization among network and data processing threads [27]. As a result, Data Shuffling in Map/Reduce-like paradigms are network-bound on relatively slow socket-based connections, when considering data-intensive workloads. Yet, they become compute bound in the presence of fast networks [28, 29].

Modern scale-out SPEs follow the so-called *shared-nothing architecture* and they are designed to execute queries in parallel on disjoint partitions of the streams. However, ingested streams may need to be partitioned according to the operator logic. Current SPEs scale out by continuously partitioning unbounded data streams into many sub-streams, on which they apply data transformations, such as stateful aggregations and joins [30]. Partitioning relies on data shuffling following message-passing and Map/Reduce-like paradigm. In this thesis, we show that this processing model does not benefit from high-speed networks and an architectural change is necessary to process streams at network speed.

Proposed solution. To enable stateful stream processing at full network bandwidth with very low latency, we introduce Slash, a novel RDMA-accelerated SPE. To this end, we propose a new system architecture to enable a processing model that omits data re-partitioning and applies stateful query logic on ingested streams. Furthermore, to deal with incoming data from a high-rate network link, we carefully manage data ingestion, scale-out processing, and consistency of distributed state to offer robust performance. Our evaluation on common streaming workloads shows that Slash outperforms baseline approaches based on data re-partitioning and is skew-agnostic. In particular, we compare Slash against a scale-out SPE (Apache Flink) on an IPoIB network, a scale-up SPE called LightSaber [5], and a self-developed straw-man solution called RDMA UpPar, which scales out query execution via RDMA-based data re-partitioning. Slash achieves up to 22x and 11.6x higher throughput than RDMA UpPar and LightSaber, respectively. Furthermore, Slash outperforms Flink by achieving an order of magnitude higher throughput on common streaming workloads. Overall, this shows that RDMA alone cannot achieve peak performance without redesigning the SPE internals.

1.2.3 State Management Runtime

The state management runtime of an SPE is responsible to store the state of each stateful operator as well as managing its life-cycle to enable fault-tolerance and resource elasticity. Current SPEs co-partition streams and state to execute stateful operators, such as aggregation and join operators, in parallel on a cluster of nodes. Each stateful pipeline of an operator maintains its own local state. Overall, the aggregated size of the state of the pipeline instances of a stateful operator

can grow to terabytes. State management is necessary when SPEs need to transparently handle faults and adjust their processing capabilities, regardless of failures or data rates fluctuations. Therefore, scale-out SPEs require efficient state management and on-the-fly reconfiguration of running queries to quickly react to spikes in the data rate or failures.

In this thesis, we show that the reconfiguration of running stateful queries in the presence of very large operator state brings a multifaceted challenge. First, a reconfiguration must have minimal impact on the performance of query processing. Second, SPEs must continuously and robustly process stream records to output correct results as if no reconfiguration ever occurred.

Problem 5: Processing and Network Overhead

A reconfiguration of a running query involves state migration between workers over a network, which results in more resource utilization and latency proportional to state size. As a result, the migration overhead induces an increment of the latency of end-to-end query processing.

To resume a stateful operator on a new node or increase the number of parallel instances of an operator, an SPE requires on-the-fly query execution plan (QEP) reconfiguration. This procedure requires two nodes to perform a handover of the state and the computation of an operator. Operators that hold large state become a bottleneck during this procedure, as the SPE needs to copy the state between two nodes. Although current SPEs provide diverse state migration techniques, they fall short in providing efficient migration for large state, as we show in this thesis.

Problem 6: Computation Consistency

A reconfiguration of a running query has to guarantee exactly-once processing semantics through consistent state management and record routing. Thus, a reconfiguration must apply changes to a running query without affecting the correctness of its results.

Migrating state to perform an on-the-fly reconfiguration of a running, distributed QEP must guarantee result correctness as well as state consistency. In particular, guaranteeing exactly-once processing is challenging, as the SPE is required to process every single record exactly-once, before, during, and after a reconfiguration.

Proposed solution. To bridge the gap between stateful stream processing and operational efficiency via on-the-fly QEP reconfigurations and state migration, we propose *Rhino*. Rhino enables on-the-fly reconfiguration of a running query to provide resource elasticity, fault tolerance, and runtime query optimizations (e.g., load balancing) in the presence of very large distributed state. In our evaluation, Rhino shows a reduction in processing latency by three orders of magnitude for a reconfiguration with large state migration. We show that Rhino does not introduce overhead on query processing, even when state is small. Furthermore, Rhino does not break exactly-once consistency guarantees, while it executes a reconfiguration. Overall, Rhino solves the multifaceted challenge of on-the-fly reconfiguration involving large (and small) stateful operators.

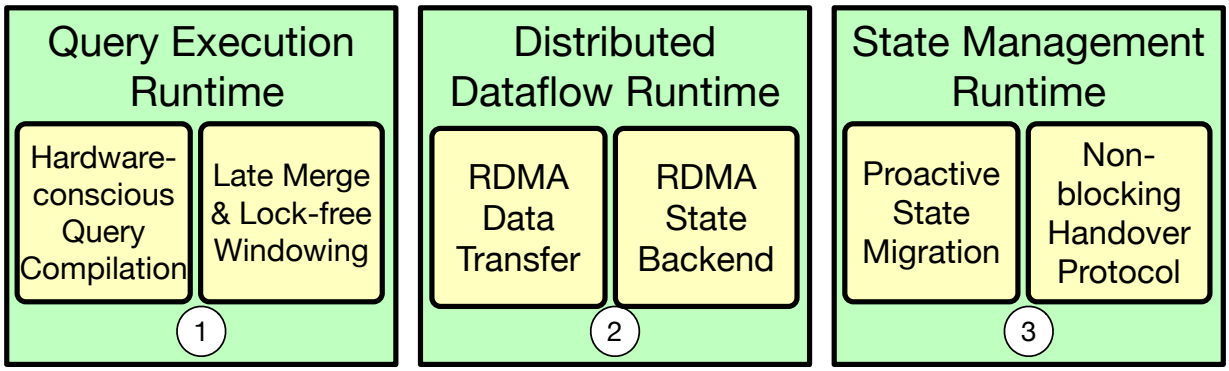


Figure 1.2: The high-level architecture of our proposed solution for high-performance SPE.

1.2.4 System Architecture

In this thesis, we devise architectural changes to improve the performance of SPEs, when they execute stateful queries. Figure 1.2 summarizes our proposed SPE architecture. First, we present a new runtime for scale-up execution of streaming queries ① to fully leverage the compute capabilities of modern CPUs. Second, we provide a novel distributed runtime for scale-out execution of streaming queries on high-speed networks ②. Finally, we investigate new techniques for a state management runtime ③ to cope with anomalous operational events, such as node failures and fluctuations in the data rate of input data streams.

Query Execution Runtime. In this thesis, we show that SPEs need to be optimized for modern hardware to exploit the possibilities of emerging hardware trends, such as multi-core processors and high-speed networks. We propose a system design that relies on compilation-based query execution, omits queue-based message passing, and uses late merging with lock-free windowing instead of partitioning. In particular, we propose a queue-less execution engine based on query compilation. This eliminates queues as a means to exchange intermediate results between operators. As a result, the proposed engine is highly suitable for an SPE on modern hardware. Furthermore, we replace partitioning with late merging strategies.

Overall, our solutions leverage the capabilities of modern hardware on a single node, efficiently. As a result, the throughput of a single node becomes sufficient for many streaming applications and can outperform a cluster of several nodes.

Distributed Dataflow Runtime. In this thesis, we show that RDMA-based acceleration of stateful streaming processing workloads is necessary to achieve robust query processing performance with high throughout and low latency. To this end, we introduce Slash: an RDMA-accelerated query execution runtime that uses RDMA-native data channels and state storage components. In particular, we design a new architecture to enable a processing model that omits data re-partitioning and applies stateful query logic on ingested streams.

Slash comprises the following building blocks: the RDMA Channel, the stateful query executor, and the Slash State Backend (SSB). First, we design an RDMA-friendly protocol to support streaming among nodes via dedicated RDMA data channels. This enables our system to execute data ingestion and data exchange among nodes at full RDMA network speed, by

leveraging the aggregated bandwidth of all NICs. Second, we devise a stateful executor that omits message passing and runs queries following late merge technique proposed in the query execution runtime. Finally, we replace re-partitioning with the SSB that enables consistent state sharing across distributed nodes. This enables multiple nodes to concurrently update the same key-value pair of the state (for instance, a group of a windowed aggregation). Overall, Slash executors outperform baseline systems as it scales out computation by eagerly applying stateful operators on data streams to compute partial state.

State Management Runtime. In this thesis, we show that runtime reconfigurations of running stateful streaming queries enable achieving robust query processing performance. To do so, we devise protocols to reconfigure running queries that comprise stateful operators with large state, to support fault-tolerance, resource elasticity, and runtime optimizations, such as load balancing. To bridge the gap between stateful stream processing and operational efficiency via on-the-fly QEP reconfigurations and state migration, we propose *Rhino*. Rhino is a library for efficient management of very large distributed state compatible with SPEs based on the streaming dataflow paradigm [15]. Rhino enables on-the-fly reconfiguration of a running query to provide resource elasticity, fault tolerance, and runtime query optimizations (e.g., load balancing) in the presence of very large distributed state. In particular, Rhino proactively migrates state so that a future reconfiguration requires minimal state transfer. Rhino applies a state-centric, proactive replication protocol to asynchronously replicate the state of a running operator on a set of SPE workers through incremental checkpoints. Furthermore, Rhino applies a handover protocol that smoothly migrates processing and state of a running operator among workers. This does not halt query execution and guarantees exactly-once processing. In contrast to state-of-the-art SPEs, our protocols are tailored for resource elasticity, fault tolerance, and runtime query optimizations.

Modularity. Note that our solutions target the runtime internals of an SPE and they do not modify the user-facing programming model of the SPE. Consequently, they are transparent to the users, who can immediately benefit from performance improvement without changing their queries. Furthermore, our contributions are modular: system builders may adopt our solutions for a specific runtime only. One use case deals with SPEs designed for the Internet-of-Things that require solutions for efficient scale-up processing, such as query compilation, to improve query processing in environments comprising heterogenous devices. An example that shows the modularity of our contributions is NebulaStream [31] (NES), which is an SPE for the Internt-of-Things developed as part of an on-going research project at TU Berlin. NES leverages the contributions of this thesis for its query processing and state management runtimes, to improve stateful query processing in the presence of heterogenous and unreliable devices. Furthermore, cloud-base SPEs, such as Apache Flink, may benefit from the integration of the solutions proposed in our work to enable resource elasticity and fault-tolerance in the presence of large operator state.

1.3 Impact of Thesis Contributions

Research Publications. The core results of this thesis resulted in the following peer-reviewed publications that have been published at international top-tier venues:

1. **Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl**: *Analyzing Efficient Stream Processing on Modern Hardware*. Proceeding of the Very Large Databases (VLDB) Endowment 12 (5), 516-530, 2019.

2. **Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, Volker Markl**: *Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines*. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020.

3. **Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, Volker Markl**: *Rethinking Stateful Stream Processing with RDMA*. Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data, 2022.

4. **Bonaventura Del Monte**: *Efficient Migration of Very Large Distributed State for Scalable Stream Processing*. Proceedings of the VLDB 2017 PhD Workshop, 2017.

In *Analyzing Efficient Stream Processing on Modern Hardware*, we contribute by investigating the design alternatives for stream processing on modern hardware as well as verifying correctness of the proposed solutions. Furthermore, we contribute by evaluating the query processing performance of the different proposed approaches and systems. In *Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines* and *Rethinking Stateful Stream Processing with RDMA*, we contribute by devising the system design and protocols, building the system artifacts, and carrying out all experiments. In *Efficient Migration of Very Large Distributed State for Scalable Stream Processing*, we contribute by proposing the research issues behind the management of very large state for stateful stream processing.

Summary. With this thesis, we lay the foundation for highly-efficient SPEs by the in-depth exploration of design alternatives and the description of a set of design changes to incorporate into current and future SPEs. We include our research contributions in the NebulaStream platform [32] to create an efficient system runtime for the execution of stateful stream processing queries with high throughput and low latency.

Furthermore, our research work provides a basis for future research. For example, future researchers may investigate state management techniques for the emerging Internet-of-Things use-cases, which involve many heterogenous, interconnected, yet failure-prone devices. In this scenario, Rhino requires further extensions to support exactly-once semantics, tight latency requirements, and volatile infrastructures.

1.4 Structure of the Thesis

The rest of this thesis is structured as follows.

Chapter 2: Background. In this Chapter, we provide the necessary background knowledge that serves as a basis for our thesis. In particular, we summarize current approaches to scale-up and scale-out stream processing, modern hardware, and state management.

Chapter 3: Analyzing Efficient Stream Processing on Modern Hardware. In this Chapter, we conduct an extensive experimental analysis of current SPEs and SPE design alternatives optimized for modern hardware. We run a series of benchmarks on common workloads and show that our hand-coded query implementations outperforms current SPEs. Driven by these findings, we propose a set of changes for future SPEs to efficiently exploit modern hardware capabilities.

Chapter 4: Rethinking Stateful Stream Processing with RDMA. In this Chapter, we describe the acceleration of stateful streaming workloads using high-speed networks that feature RDMA. We propose a set of architectural changes to current SPE to natively integrate RDMA and implement them in a research prototype called Slash. We validate our design via a series of experiments and show that Slash outperforms current SPEs.

Chapter 5: Efficient Management of Very Large Distributed State for Stream Processing Engines. In this Chapter, we research state management techniques to enable efficient stateful stream processing in the presence of failures and fluctuating data rates. Guaranteeing exactly-once processing semantics and tight latency requirements of industrial setups are challenged by large operator state that induces challenges, such as network overhead. To meet industrial needs, we present Rhino: a system library that provides an efficient state management runtime to cope with failures and data rate fluctuations, even in the presence of large operator state.

Chapter 6: Additional Contributions. In this Chapter, we describe further related research contributions, which have been made while working on this thesis, but are not covered in other chapters.

Chapter 7: Conclusion. In this Chapter, we conclude the thesis by summarizing our contributions and providing an outlook to future work.

2

Background

In this Chapter, we provide the necessary background knowledge that serves as a basis for our thesis: data stream processing and modern hardware. In the first half, we describe stateful streaming processing 2.1, current systems 2.2, and state management techniques for SPEs 2.3. In the second half, we present an overview of query processing on modern hardware with a focus on RDMA.

2.1 Stateful Stream Processing

Modern scale-out SPEs leverage the dataflow execution model [33] to run continuous queries. This system design enables the execution of queries on a cluster of servers. A query consists of stateful and stateless operators [14, 15]. An operator is either stateless or stateful. The output of a stateful operator is not only determined by the input but also by intermediate state. A QEP is represented as a weakly-connected graph with special vertices called source and sink operators. Sources allow for stream data ingestion, whereas sinks output results to external systems. To process data, users define transformations through higher-order functions and first-order functions (UDFs). Operators and UDFs support internal, mutable state (e.g., windows, counters, and ML models).

In the rest of this Chapter, we follow the definitions introduced by Fernandez et al. [34] to model streaming processing. In this model, a query is a set of logical operators. Each logical operator o consists of p_o physical instances $o_1, ..., o_i, ..., o_{p_o}$. A stream represents an infinite set of records $r = (k, t, a)$, where $k \in K$ is a partitioning key, K is the key space, t is a strictly monotonically increasing timestamp, and a is a set of attributes. An operator produces and consumes n and m streams, respectively.

To execute a query, an SPE maps the parallel instances of operators to a set of worker servers. Each operator has its producer operators (*upstream*) and its consumer operators (*downstream*).

If two operators are in a producer-consumer relation, they have a pattern for data exchange, e.g., hash-partitioning, round-robin, or broadcast. A parallel instance receives records from upstream operators through channels according to an exchange pattern, e.g., hash-partitioning. An inter-operator channel is durable, bounded, and guarantees FIFO delivery. A channel contains fixed-sized buffers, which store records of variable size and control events, e.g., watermarks [15]. A channel imposes a total order on its records and control events based on their timestamp, however, instances with multiple channels are non-deterministic. As a result, there exists a partial order among records consumed by an instance.

A stateful operator o holds its state S_o, which is a mutable data set of key-value pairs (k, v). To scale out, an SPE divides S_o in disjoint partitions and assigns a partition $S^t_{o_i}$ to an instance o_i using k as partitioning key. With t, we denote the timestamp of the last update of a state partition. A value v of the state is an arbitrary data type. A parallel instance o_i processes every record in a buffer, reacts to control events, emits records, and reads or updates its state $S^t_{o_i}$.

2.2 Streaming Processing Engines

Over the last decades, two categories of streaming systems emerged: *scale-out* SPEs and *scale-up* SPEs. SPEs in the first category are optimized for *scale-out* execution of streaming queries on shared-nothing architectures. In general, these SPEs apply a distributed producer-consumer pattern and a buffer mechanism to handle data shuffling among operators (e.g., partitioning). Their goal is to massively parallelize the workload among many small to medium sized nodes. Example of scale-out SPEs are Flink [35], Storm [22], Spark Streaming [36], Millwheel [37], Google Dataflow [15], and TimelyDataflow [21], which parallelize queries on shared-nothing architectures. In this thesis, we analyze Apache Flink [20], Apache Spark [2], and Apache Storm [22] as representative, JVM-based, state-of-the-art scale-out SPEs. We choose these SPEs due to their maturity, wide academic and industrial adoption, and the size of their open-source communities.

SPEs in the second category are optimized for *scale-up* execution on a single machine. Their goal is to exploit the capabilities of one high-end machine efficiently. Recent SPEs in this category are Streambox [4], LightSaber [5], BriskStream [6], Grizzly [38], Trill [39], and Saber [40]. In particular, Streambox aims to optimize the execution for multi-core machines, whereas SABER takes heterogeneous processing into account by utilizing GPUs. Finally, Trill supports a broad range of queries beyond SQL-like streaming queries. In this thesis, we examine Streambox, LightSaber, and SABER as representative SPEs in this category.

A major aspect of parallel stream processing engines are the underlying processing models which are either based on micro-batching or use pipelined tuple-at-a-time processing [15, 41, 33, 42, 43]. *Micro-batching* SPEs split streams into finite chunks of data (batches) and process these batches in parallel. SPEs such as Spark [36], Trident [44], and SABER [40] adopt this micro-batching approach. In contrast, *pipelined, tuple-at-a-time* systems execute data-parallel pipelines consisting of stream transformations. Instead of splitting streams into micro-batches, operators

receive individual tuples and produce output tuples continuously. SPEs such as Apache Flink [45] and Storm [22] adopt this approach.

Scale-up and scale-out SPEs assume common data and query models, yet they execute queries differently. We summarize their data, query, and processing models in the following.

Data and query model. We follow the definitions introduced by Fernandez et al. [34] and assume a data stream to consist of an immutable, unbounded set of records. A record contains a timestamp t, a primary key k, and a set of attributes. Timestamp are strict monotonically increasing and used for windowing related operations as well as progress tracking. Streaming queries are modelled as directed acyclic graphs with stateful operators as vertices and data flows as edges. The output of a streaming operator depends on content, timestamp or arrival order of input records, and its intermediate state. In general, an operator must output no result at a timestamp t that is computed using records bearing timestamps greater than t.

Scale-up execution. Scale-up SPEs rely on task-based parallelization, compilation-based operator fusion [46], and late merge [47] to fully utilize available hardware resources. Logical operators are fused together and compiled to machine code, which the SPE executes on inbound data buffers using task-based parallelization. Tasks may concurrently update a global operator state or eagerly update local state, which the SPE eventually merges and ensure its consistency.

Scale-out execution. Scale-out SPEs use operator-to-thread parallelism and data re-partitioning [46] to scale out. Each logical operator consists of p physical operators, which the SPE runs in parallel on a cluster of nodes. Scale-out SPEs re-partition input streams so that each physical operators applies stateful transformations on a disjoint partition of the data. This enables consistent stateful computations via local mutable state [30]. Parallel instances receive records from upstream operators via in-memory or network-based *data channels* following an exchange pattern.

2.3 State Management in SPEs

Today's SPEs provide state management through two techniques: *state checkpointing* and *state migration*.

State Checkpointing. State checkpointing enables an SPE to consistently persist the state of each operator, recover from failures, and reconfigure a running query.

Fernandez et al. propose a checkpointing approach [34, 34] that stores operator state and in-flight records. This produces local checkpoints that are later used to reconfigure or recover operators. In contrast, Carbone et al. [1] propose a distributed protocol for global, epoch-consistent checkpoints. It captures only operator state (by skipping in-flight records) and does not halt query processing. This protocol divides the execution of a stateful query into *epochs*. An epoch is a set of consecutive records that the SPE processes in a time frame. If query execution fails before completing the $i + 1$-th epoch, the SPE rollbacks to the state of the i-th epoch.

The distributed protocol of Carbone et al. [1] assumes reliable data channels and FIFO delivery semantics. Thus, each channel ensures in-order, consistent delivery of records to a downstream

instance. In addition, it assumes upstream backups of ingested streams, i.e., sources can consume records again from an external system. Finally, it assumes checkpoints to be persisted on a distributed file system for availability. The drawback of this protocol is the full restart of a query upon a reconfiguration.

State Migration. State migration enables the handover of the processing and the state of a key partition of an operator among workers [48]. This enables on-the-fly query reconfiguration to support fault-tolerance, resource elasticity, and runtime optimizations.

Megaphone is the latest approach to state migration [48]. Megaphone is a state migration technique built on Timely Dataflow [21] that enables consistent reconfiguration of a running query through fine-grained, fluid state migration. Megaphone introduces two migrator operators for each migratable operator in a QEP to route records and state according to a planned migration. It requires from an SPE: 1) out-of-band progress tracking, i.e., operators need to observe each other's progress and 2) upstream operators need to access downstream state. Its authors state that Megaphone can be compatible with other SPEs but with some overhead.

2.4 Modern Hardware and Stream Processing Engines

Any data management system - including SPEs - designed for modern hardware has to exploit three critical hardware resources efficiently to achieve a high resource utilization: CPU, main memory, and network. Current SPEs are designed around the assumption of commodity hardware that was prominent in private and public clouds [49]. However, datacenter technology has evolved in the past decades, resulting in the availability of more powerful compute resources, faster and larger memory resources, as well as faster network resources. In this thesis, we investigate if and how current SPEs and their system designs exploit these resources efficiently. We first provide an overview of new compute and memory resource in Section 2.4.1. After that, we describe advancements in datacenter networking technology in Section 2.4.2.

2.4.1 Compute and Main-Memory

Current datacenters contain servers with large memory and compute capabilities. On the memory side, today's common servers provide *main memory* capacities of several terabytes. As a result, the majority of the data and state information, such as the data of one window, can be stored in main memory. In combination with multiple threads working on the same data, this significantly speeds up the processing compared to maintaining the same state on slower disk or on remote servers. However, a streaming system that uses main-memory has to handle synchronization and parallel aspects in much shorter time frames to enable efficient processing.

On the compute side, modern *many-core CPUs* contain dozens of CPU cores per socket. Additionally, multi-socket CPUs connect multiple CPUs via fast interconnects to form a network of cores. However, this architecture introduces non-uniform memory access (NUMA) among cores [50, 51]. As a result, the placement of data in memory directly affects the overall efficiency of the

system. Furthermore, each core caches parts of the data in its private and shared caches for fast access and consists - on a high-level - of two pipelined components: a *front-end* and a *back-end*.

In the following, we provide a concise overview of the CPU micro-architectures, including cache hierarchy design. Caches are organized hierarchically and sits in front of main-memory to reduce memory-to-register data movement. Modern CPU architecture, such as Intel/AMD x86 as well as ARM, include three levels of caches: two private levels of caches, i.e., L1 and L2, and a shared L3 level of cache. The number of clock cycles required to access a cache is increased as the level of the cache is increased in the memory hierarchy. Moreover, the size of a cache in bytes is increased as the level of the cache is increased in the memory hierarchy. The L3 cache is connected to main-memory and is shared among all cores, whereas L1 and L2 are private to specific cores. We distinguish two types of L1 cache: the L1d (data) cache and the L1i (instruction) cache, which store data and instructions, respectively.

The front-end of a core decodes instructions stored in the L1i cache into *μ-ops* and delivers up to four μ-ops per cycle to the back-end. The back-end processes μ-ops out-of-order by allocating execution units and loading data from memory. Completed μ-ops are defined as *retired* (R) and constitute the useful work performed by a CPU. If the front-end stalls, the rename/allocate part of the out-of order engine starves and thus execution becomes front-end bound. Second, the *back-end* processes instructions issued by the front-end. If the back-end stalls because all processing resources are occupied, the execution becomes back-end bound. Furthermore, we divide back-end stalls into stalls related to the memory sub-system (called *Memory bound*) and stalls related to the execution units (called *Core bound*). Third, *bad speculation* summarizes the time that the pipeline executes speculative micro-ops that never successfully retire. This time represents the amount of work that is wasted by branch mis-predictions. Fourth, *retiring* refers to the number of cycles that are actually used to execute useful instructions. This time constitutes the amount of useful work done by the CPU.

Overall, to exploit this performance critical resource efficiently, system builders need to take into account the data and instruction locality as well as the micro-architectural intricacies of modern CPUs.

2.4.2 Networking

Over the past years, datacenter network technology has become faster and has finally outperformed or reached the same speed of main memory bandwidth [28]. In fact, commonly available Ethernet technologies provide 1, 10, 40, or 100 Gbit bandwidth. In contrast, new network technologies, such as InfiniBand or Converged Ethernet, provide much higher bandwidth up to or even faster than main memory bandwidth [28]. This important trend will lead to radical changes in system designs. In particular, the common wisdom of processing data *locally first* does not hold with future network technologies [28]. As a result, a system which transfers data between nodes as fast as or even faster than reading from main memory introduces new challenges and opportunities to future SPEs.

One important network feature that enabled the significant increment of network capabilities is RDMA. RDMA is a communication stack provided by Infiniband (IB), RoCE (RDMA over Converged Ethernet), and iWarp (Internet Wide Area RDMA Protocol) networks [52]. RDMA enables access to the main memory of a remote node with minimal involvement of the remote CPU. As a result, RDMA achieves high bandwidth (up to 200 Gbps per port [53]) and low latency (up to 2µs per round-trip [52]). RDMA offers bidirectional data transfer via *zero-copy*, which bypasses the kernel network stack. In contrast, socket-based protocols, such as TCP, involve costly system calls and data copies between user- and kernel-space [28]. RDMA-capable NICs also support socket-based communication via IP-over-InfiniBand (IPoIB). However, this approach results in lower efficiency [28]. RDMA provides two APIs (so-called *verbs*) for communication: one-sided and two-sided verbs APIs [52]. Besides, RDMA provides *reliable*, *unreliable*, and *datagram* connections. A reliable connection enables one-sided verbs and in-order packet delivery, while unreliable and datagram connections may drop packets. Reliable and unreliable connections allow unicast connections between two endpoints. In contrast, datagram connections allow a local QP to engage multiple remote endpoints. With two-sided verbs (Send-Recv), sender and receiver are actively involved in the communication. The receiver polls for incoming message, which requires CPU involvement. In contrast, one-sided verbs involves one active sender and one passive receiver (RDMA WRITE) or a passive sender and an active receiver (RDMA READ). They enable more efficient data transfer, but need synchronization to detect inbound messages. Besides, one-sided verbs provide atomic semantics, such as `compare-and-swap`, for 8-byte values [28].

RDMA requires explicit memory management and asynchronous interaction between CPU and NIC. In the following, we explain the lifecycle of an RDMA connection (from its setup to runtime operations).

1) Sender and receiver establish communication via a *queue pair* (QP), which consists of a send and a receive queue, and a *completion queue* (CQ).
2) Applications need to register in advance RDMA-capable memory to enable *direct memory access* (DMA) operations operations. Thereby, sender and receiver must know the memory locations of its counterpart (one-sided verbs) or state the intention to receive a buffer (two-sided verbs)
3) A sender pushes *work queue elements* (WQEs) into a QP, which contain verb and communication parameters.
4) The sender NIC asynchronously reads memory chunks for each WQE using DMA and sends them to a receiver NIC as packets. Thus, the sender must not alter the content of the local memory while transfer is in progress.
5) For every inbound packet, the receiver NIC issues writes to main memory via DMA. Thus, an RDMA transfer neither consumes CPU resources nor pollutes CPU caches of the remote machine.
6) Upon the transmission of the last packet, the sender NIC pushes a completion event in its local CQ.

RDMA provides two major benefits: it 1) enables fast data transfer and 2) shares memory areas among nodes [54]. However, RDMA-enabled systems need careful design, as RDMA does not offer coherence between local and remote memory. Instead, this is offloaded to the application. Besides, coherence among NIC memory, main memory, and the CPU is vendor-dependent [28]. The choice of verbs and parameters, such as message size, is application-sensitive and requires careful tuning [52].

3

Analyzing Efficient Stream Processing on Modern Hardware

3.1 Introduction

Over the last decade, streaming applications have emerged as an important new class of big data processing use cases. Streaming systems have to process high velocity data streams under tight latency constraints. To handle high velocity streams, modern SPEs such as Apache Flink [20], Apache Spark [2], and Apache Storm [22] distribute processing over a large number of computing nodes in a cluster, i.e., *scale-out* the processing. These systems trade single node performance for scalability to large clusters and use a Java Virtual Machine (JVM) as the underlying processing environment for platform independence. While JVMs provide a high level of abstraction from the underlying hardware, they cannot easily provide efficient data access due to processing overheads induced by data (de-)serialization, objects scattering in main memory, virtual functions, and garbage collection. As a result, the overall performance of scale-out SPEs building on top of a JVM are severely limited in throughput and latency. Another class of SPEs optimize execution to *scale-up* the processing on a single node. For example, Saber [40], which is build with Java, Streambox [4], which is build with C++, and Trill [39], which is build with C#.

In Figure 3.1, we show the throughput of modern SPEs executing the Yahoo! Streaming Benchmark [55] on a single node compared to hand-coded implementations. We report the detailed experimental setup in Section 3.4.1. As shown, the state-of-the-art SPEs Apache Flink, Spark, and Storm achieve sub-optimal performance compared to the physical limit that is established by the memory bandwidth. To identify bottlenecks of SPEs, we hand-code the benchmark in Java and C++ (HC bars). Our hand-coded Java implementation is faster than Flink but still 40 times slower than the physical limit. In contrast, our C++ implementation and Streambox omit serialization, virtual functions, and garbage collection overheads and store

tuples in dense arrays (arrays of structures). This translates to a 26 times better performance for our C++ implementation and a 6 times better performance for Streambox compared to the Java implementation. Finally, we show an *optimized* implementation that eliminates the potential SPE-bottlenecks that we identify in this Chapter. Our optimized implementation operates near the physical memory limit and utilizes modern hardware efficiently.

The observed sub-optimal single node performance of current SPEs requires a large cluster to achieve the same performance as a single node system using the scale-up optimizations that we evaluate in this thesis. The major advantage of a scale-up system is the avoidance of inter-node data transfer and a reduced synchronization overhead. With hundreds of cores and terabytes of main memory available, modern scale-up servers provide an interesting alternative to process high-volume data streams with high throughput and low latency. Utilizing the capabilities of modern hardware on a single node efficiently, we show that the throughput of a single node becomes sufficient for many streaming applications and can outperform a cluster of several nodes. As a result, we show that SPEs need to be optimized for modern hardware to exploit the possibilities of emerging hardware trends, such as multi-core processors and high-speed networks.

We investigate streaming optimizations by examining different aspects of stream processing regarding their exploitation of modern hardware. To this end, we adopt the terminology introduced by Hirzel et al. [25] and structure our analysis in data-related and processing-related optimizations. As a result, we show that by applying appropriate optimizations, single node throughput can be increased by two orders of magnitude. Our contributions are as follows:

- We analyze current SPEs and identify their inefficiencies and bottlenecks on modern hardware setups.
- We explore and evaluate new architectures of SPEs on modern hardware that address these inefficiencies.
- Based on our analysis, we describe a set of design changes to the common architecture of SPEs to *scale-up* on modern hardware.
- We conduct an extensive experimental evaluation using the Yahoo! Streaming Benchmark, the Linear Road Benchmark, and a query on the NY taxi data set.

The rest of this Chapter is structured as follows. We explore the data-related (see Section 3.2) and processing-related stream processing optimization (see Section 3.3) of an SPE on modern hardware. In Section 3.4, we evaluate state-of-the-art SPEs and investigate different optimizations in an experimental analysis. Finally, we discuss our results in Section 3.4.3 and present related work in Section 3.5.

3.2 Data-Related Optimizations

In this section, we explore data-related aspects for streaming systems. We discuss different methods to supply high velocity streams in Section 3.2.1 and evaluate strategies for efficient data passing among operators in Section 3.2.2.

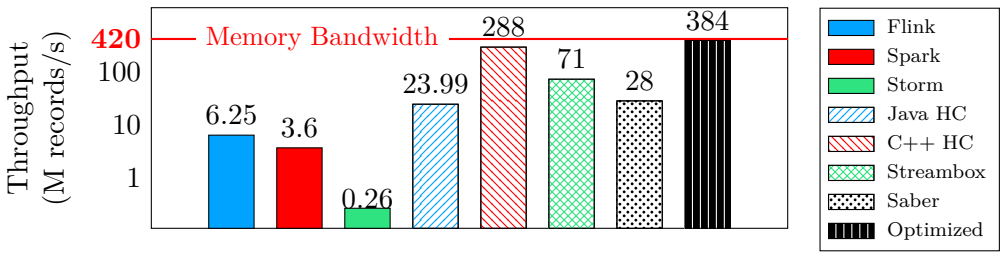


Figure 3.1: Yahoo! Streaming Benchmark (1 Node).

3.2.1 Data Ingestion

We examine different alternatives how SPEs can receive input streams. In Figure 3.2, we provide an overview of common ingestion sources and their respective maximum bandwidths. In general, an SPE receives data from network, e.g., via sockets, distributed file systems, or messaging systems. Common network bandwidths range from 1 to 100 GBit over Ethernet (125 MB/s - 12.5 GB/s). In contrast, InfiniBand (IB) offers higher bandwidths from 1.7 GB/s (FDR 1x) to 37.5 GB/s (EDR - Enhanced Data Rate - 12x) per network interface controller (NIC) per port [56].

3.2.1.1 Analysis

Based on the numbers in Figure 3.2, we conclude that modern network technologies enable ingestion rates near main memory bandwidth or even higher [28]. This is in line with Binning et al. who predict the end of slow networks [28]. Furthermore, they point out that future InfiniBand standards such as HDR or NDR will offer even higher bandwidths. Trivedi et al. [57] assess the performance-wise importance of the network for modern distributed data processing systems such as Spark and Flink. They showed that increasing the network bandwidth from 10 Gbit to 40 Gbit transforms those systems from network-bound to CPU-bound systems. As a result, improving the single node efficiency is crucial for scale-out systems as well. On single node SPEs, Zhang et al. [16] point out that current SPEs are significantly CPU-bound and thus will not natively benefit from an increased ingestion rate. An SPE on modern hardware should be able to exploit the boosted ingestion rate to increase its overall, real-world throughput.

3.2.1.2 Infiniband and RDMA

The trends in the area of networking technologies over the last decade showed, that Remote Direct Memory Access (RDMA) capable networks became affordable and thus are present in an increasing number of data centers [26]. RDMA enables the direct access to main memory of a remote machine while bypassing the remote CPU. Thus, it does not consume CPU resources of the remote machine. Furthermore, the caches of the remote CPU are not polluted with the transfered memory content. Finally, *zero-copy* enables direct send and receive using buffers and bypasses the software network stack. Alternatively, high-speed networks based on Infiniband can

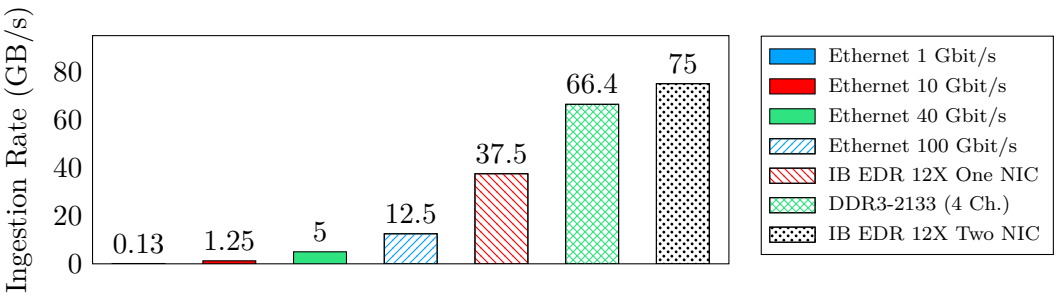


Figure 3.2: Data ingestion rate overview.

run common network protocols such as TCP and UDP (e.g., via IPoIB). However, this removes the benefits of RDMA because their socket interfaces involve system calls and data copies between application buffers and socket buffers [58].

Today, three different network technologies provide RDMA-support: 1) InfiniBand, 2) RoCE (RDMA over Converged Ethernet), and 3) iWARP (Internet Wide Area RDMA Protocol). The underlying NICs are able to provide up to 100 Gbps of per-port bandwidth and a round-trip latency of roughly 2μs [26]. Overall, there are two modes for RDMA communication: one-sided and two-sided. With two-sided communication, both sides, the sender and the receiver are involved in the communication. Thus, each message has to be acknowledged by the receiver and both participants are aware that a communication takes place. In contrast, one-sided communication involves one active sender and one passive receiver (RDMA write) or a passive sender and an active receiver (RDMA read). In both cases, one side is agnostic of the communication. As a result, synchronization methods are required to detect that a communication is finished.

3.2.1.3 Experiment

In Figure 3.3, we run the aforementioned communication modes on an InfiniBand high-speed network with a maximum bandwidth of 56 Gbit/s (FDR 4x). On the x-axis, we scale the buffer/message and on the y-axis we show the throughput in GBytes/s for a data transfer of 10M tuples of 78 byte each, i.e., the yahoo streaming benchmark tuples. To implement the C++ version, we use RDMA Verbs [59]. The Java implementation uses the *disni* library [60] and follows the Spark RDMA implementation [61]. As a first observation, Figure 3.3 shows that all modes perform better with larger buffer/messages sizes and their throughput levels off starting around 1M. Second, the C++ RDMA read implementation achieves roughly 2 GB/s more throughput than the Java implementation. As a result, we conclude that the JVM introduces additional overhead and, thus, a JVM-based SPE is not able to utilize the entire network bandwidth of modern high-speed networks. Third, the C++ RDMA write operation is slightly slower than the read operation. Finally, the two-sided send/receive mode is slower than the one-sided communication modes. Overall, the one-sided RDMA read communication using a message size of 4 MB performs best. Note that, our findings are in line with previous work [62, 26].

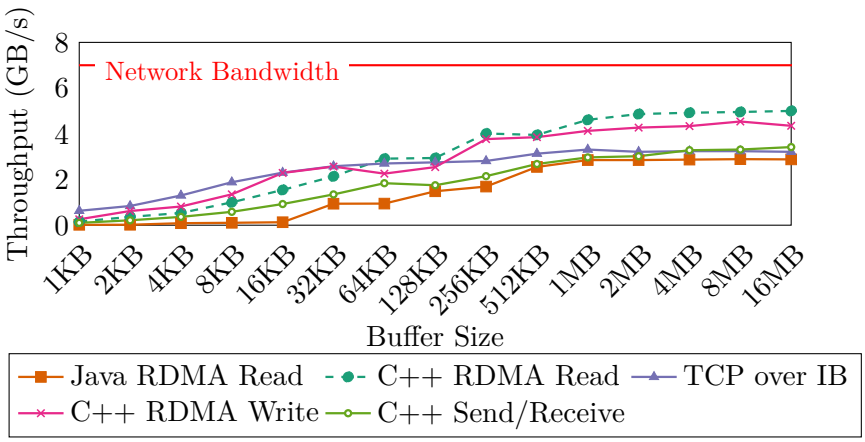


Figure 3.3: Infiniband network bandwidth.

3.2.1.4 Discussion

Our experiments show that a JVM-based distributed implementation is restricted by the JVM and cannot utilize the bandwidth as efficiently as a C++ implementation. As a result, the performance of JVM-based SPEs is limited in regards to upcoming network technologies that are able to deliver data at memory-speed.

3.2.2 Data Exchange between Operators

In this section, we examine the design of pipelined execution using message passing, which is common in state-of-the-art SPEs. In particular, SPEs execute data-parallel pipelines using asynchronous message passing to provide high-throughput processing with low-latency. To pass data among operators, they use queues as a core component. In general, data is partitioned by keys and distributed among parallel instances of operators. In this case, queues enable asynchronous processing by decoupling producer and consumer. However, queues become the bottleneck if operators overload them. In particular, a slow consumer potentially causes back-pressure and slows down previous operators. To examine queues as the central building block in modern SPEs, we conduct an experiment to asses the capabilities of open-source, state-of-the-art queue implementations.

3.2.2.1 Experimental Setup

In this experiment, we selected queues that cover different design aspects such as in memory management, synchronization, and underlying data structures. We select queues provided by Facebook (called Folly) [63], the Boost library [64], the Intel TBB library [65], and the Java *java.util.concurrent* package [66]. In addition, we choose an implementation called *Moody* [67] and an implementation based on memory fences [68]. Finally, we provide our own implementations based on the STL queue and list templates. Except the Java queue, all queues are implemented in C++.

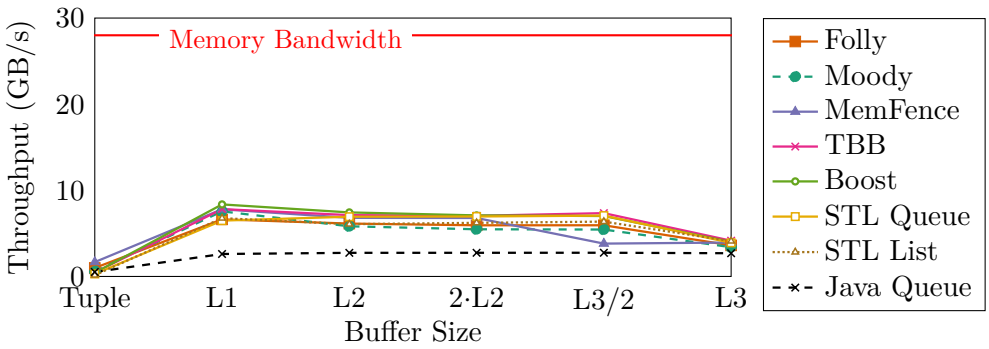


Figure 3.4: Queue throughput.

Following Sax et al. [69] and Carbone et al. [20], we group tuples into buffers to increase the overall throughput. They point out, that buffers introduce a trade-off between latency and throughput. For our C++ implementations, we enqueue filled buffers into the queue by an explicit copy and dequeue the buffer with a different thread. In contrast, our Java queue implementation follows Flink by using buffers as an unmanaged memory area. A tuple matches the size of a cache line, i.e., 64 Byte. We pass 16.7M tuples, i.e., 1 gigabyte of data, via the queue.

3.2.2.2 Observation

In Figure 3.4, we relate the throughput measurements in GB/s to the physically possible main memory bandwidth of 28 GB/s. Furthermore, we scale the *buffer size* on the x-axis such that it fits a particular cache size. In general, the lock-free queues from the TBB and Boost library achieve the highest throughputs of up to 7 GB/s, which corresponds to 25% of the memory bandwidth. Other C++ implementations achieve slightly lower throughput of up to 7 GB/s. In contrast, the Java queue, which uses conditional variables, achieves the lowest throughput of up to 3 GB/s. Overall, buffering improves throughput as long as the buffer size either matches a private cache size (L1 or L2) or is smaller than L3 cache size. Inside this range, all buffer sizes perform similar with a small advantage for a buffer size that matches the L1 cache.

3.2.2.3 Discussion

Figure 3.4 shows that queues are a potential bottleneck for a pipelined streaming system using message passing. Compared to a common memory bandwidth of approximately 28 GB/s, the best state-of-the-art queue implementations exploit only one fourth of this bandwidth. Furthermore, a Java queue achieves only 10% of the available memory bandwidth because it exploits conditional variables and unmanaged buffers. These design aspects introduce synchronization as well as serialization overhead.

As a result, the maximum throughput of a streaming system using queues is bounded by the number of queues and their aggregated maximum throughput. Additionally, load imbalances reduce the maximum throughput of the entire streaming system. In particular, a highly skewed data stream can overload some queues and under-utilize others.

3.2.3 Discussion

The results in Figure 3.3, suggest to use RDMA read operations and a buffer size of several megabytes to utilize todays high-speed networks in a distributed SPE. The results in Figure 3.4 strongly suggest that a high performance SPE on modern hardware should omit queues, as they induce a potential bottleneck. If queues are used, the buffer size should be between the size of the L1 and twice the L2 cache, based on our measurements.

3.3 Processing-related Optimizations

In this section, we examine the processing-related design aspects of SPEs on modern hardware. Following the classification of stream processing optimizations introduced by Hirzel et al. [25], we address operator fusion in Section 3.3.1, operator fission in Section 3.3.2, and windowing techniques in Section 3.3.3. We will address the stream-processing optimization of *placement* and *load balancing* in the context of operator fusion and fission [25].

3.3.1 Operator Fusion

We examine two different execution models for SPEs, namely *query interpretation* and *compilation-based query execution*. Both models enable data-parallel pipelined stream processing. For our following considerations, we assume a query plan that is translated into a processing pipeline consisting of sources, intermediate operators, and sinks. Sources continuously provide input data to the processing pipeline and sinks write out the result stream. In general, we can assign a different degree of parallelism to each operator as well as to each source and sink.

3.3.1.1 Interpretation-based Query Execution

An interpretation-based processing model is characterized by: 1) an *interpretation-based evaluation* of the query plan, 2) the *application of queues* for data exchange, and 3) the possibility of *operator fusion*. In Figure 3.5a, we show an example query plan of a simple select-project-aggregate query using a query interpretation model. To this end, a query plan is translated into a set of pipelines containing producer-consumer operators. Each operator has its own processing function which is executed for each tuple. The interpretation-based processing model uses queues to forward intermediate results to downstream operators in the pipeline. Each operator can have multiple instances where each instance reads from an input queue, processes a tuple, and pushes its result tuples to its output queue. Although all operators process a tuple at a time on a logical level, the underlying engine might perform data exchanges through buffers (as shown in Section 3.2.2).

In general, interpretation-based processing is used by modern SPEs, such as Flink, Spark, and Storm to link parallel operator instances using a push-based approach implemented by function calls and queues. In case an operator instance forwards its results to only one subsequent operator, both operators can be fused. Operator fusion combines operators into a single tight *forloop* that

passes tuples via register and thus eliminating function calls. Note, that this register-level operator fusion goes beyond function call-based operator fusion used in other SPS [70, 3, 71]. However, many operators, such as the aggregate-by-key operator in Figure 3.5a, require a data exchange due to partitioning. In particular, each parallel instance of the aggregate-by-key operator processes a range of keys and stores intermediate results as internal operator state. As a result, queues are a central component in an interpretation-based model to link operators. Note that, this execution model represents each part of the pipeline as a logical operator. To execute a query, every logical operator is mapped to a number of physical operators, which is specified by the assigned degree of parallelism.

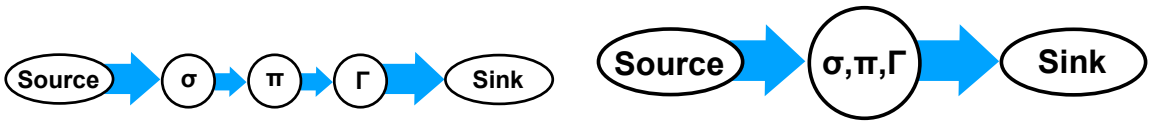
3.3.1.2 Compilation-based Query Execution

A compilation-based processing model follows the ideas of Neumann [24] and is characterized by: 1) the execution of compiled query plans (using custom code generation), 2) operator fusion performed at query compile time, and 3) an improved hardware resource utilization. In general, this approach fuses operators within a pipeline until a so-called *pipeline-breaker* is reached. A pipeline breaker requires the full materialization of its result before the next operator can start processing. In the context of databases and in-memory batch processing, pipeline breakers are relational operators such as sorting, join, or aggregations. In the context of streaming systems, the notion of a pipeline breaker is different because streaming data is unbounded and, thus, the materialization of a full stream is impracticable. Therefore, we define operators that send partially materialized data to the downstream operator as *soft* pipeline breakers. With *partially materialized*, we refer to operators that require buffering before they can emit their results, e.g., windowed aggregation. Compilation-based SPEs, such as System S/SPADE and SABER, use operator fusion to remove unnecessary data movement via queues.

In Figure 3.5b, we present a select-project-aggregate query in a compilation-based model. In contrast to an interpretation-based model, all three operators are fused in a single operator that can run with any degree of parallelism. This model omits queues for message passing and passes tuples via CPU registers. Additionally, it creates a tight loop over the input stream, which increases data and instruction locality. Neumann showed that this model leads to improved resource utilizations on modern CPUs in databases [24]. In terms of parallelization, operator fusion results in a reduced number of parallel operators compared to the query interpretation-based model. Instead of assigning one thread to the producer and another thread to the consumer, this model executes both sides in one thread. Thus, with the same number of available threads, the parallelism increases.

3.3.1.3 Discussion

We contrast two possible processing models for an SPE. On a conceptual level, we conclude that the compilation-based execution model is superior for exploiting the resource capabilities of modern hardware. In particular, it omits queues, avoids function calls, and results in code



(a) Interpretation-based. **(b)** Compilation-based.

Figure 3.5: Query execution strategies.

optimized for the underlying hardware. We expect the same improvement for streaming queries as Neumann reported for database queries [24]. However, a scale-out optimized SPE commonly introduces queues to distribute the computation among different nodes to decouple the producer from the consumer.

3.3.2 Operator Fission

In this section, we present two general parallelization strategies to distribute the processing among different computing units. This stream processing optimization includes partitioning, data parallelism, and replication [25]. With *Upfront Partitioning* we refer to a strategy that physically partitions tuples such that each consumer processes a distinct sub-set of the stream. With *Late Merging* we refer to a strategy that logically distributes chunks of the stream to consumers, e.g., using a round-robin. This strategy may require a final merge step at the end of the processing if operators such as keyed aggregations or joins are used.

3.3.2.1 Upfront Partitioning.

Upfront Partitioning introduces a physical partitioning step between producer and consumer operators. This partitioning consists of an intermediate data shuffling among n producers and m consumers, which are mapped to threads. Each producer applies a partitioning function (e.g., consistent hashing or round-robin) after processing a tuple to determine the responsible consumer. In particular, partitioning functions ensure that tuples with the same key are distributed to the same consumer. In Figure 3.6a, we present this common partition-based parallelization strategy, which is used by Apache Flink [20], Apache Spark Streaming [2], and Apache Storm [22].

This strategy leverages queues to exchange data among producer and consumer operators. Thus, each producer explicitly pushes its output tuples to the input queue of its responsible consumer. Although the use of buffers improves the throughput of this strategy (see Section 3.2.2), we showed that queues establish a severe bottleneck in a scale-up SPE. Furthermore, the load balancing on consumer side highly depends on the quality of the partitioning function. If data are evenly distributed, the entire system achieves high throughput. However, an uneven distribution leads to over-/underutilized parallel instances of an operator.

The processing using upfront partitioning creates independent/distinct, thread-local intermediate results on each consumer. Thus, assembling the final output result requires only a

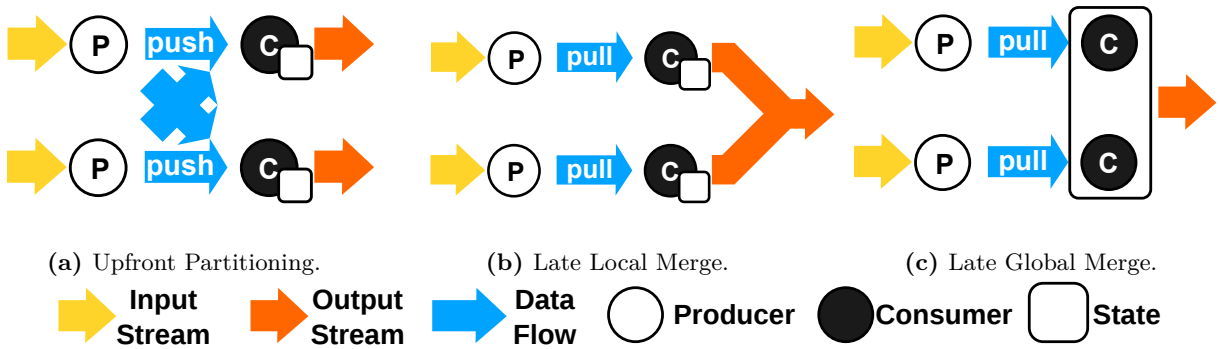


Figure 3.6: Parallelization strategies.

concatenation of thread local intermediate results. An SPE can perform this operation in parallel, e.g., by writing to a distributed file system or to a distributed message broker.

3.3.2.2 Late Merging

Late Merging divides the processing among independent threads that pull data by their own. Leis et al. [50] and Pandis et al. [72] previously applied this partitioning approach in the context of high-performance, main memory databases. In this Chapter, we revise those techniques in the context of stream processing. In contrast to upfront partitioning, this strategy requires an additional merge step at the end of the processing pipeline. In the following, we introduce two merging strategies: *Late Local Merge* and *Late Global Merge*.

In the **Late Local Merge (LM)** strategy, each worker thread buffers its partial results in a *local* data structure (see Figure 3.6b). When the processing pipeline reaches a soft pipeline-breaker (e.g., a windowed aggregation), those partial results have to be merged into a global data structure. Because each worker potentially has tuples of the entire domain, the merge step is more complex compared to the simple concatenation used by UP. One solution for merging the results of a soft pipeline-breaker such as an aggregation operator is to use parallel tree aggregation algorithms.

In the **Late Global Merge (GM)** strategy, all worker threads collaboratively create a global result during execution (see Figure 3.6c). In contrast to late local merge, this strategy omits an additional merging step at the end. Instead, this strategy introduces additional synchronization overhead during run-time. In particular, if the query plan contains stateful operators, e.g., keyed aggregation, the system has to maintain a global state among all stateful operators of a pipeline. To minimize the maintenance costs, a lock-free data structure should be utilized. Furthermore, contention can be mitigated among instances of stateful operators through fine-grained updates using atomic compare-and-swap instructions. Finally, Leis et al. [50] describe a similar technique in the context of batch processing, whereas Fernandez et al. [73] provide solutions for scale-out SPEs.

Depending on the characteristics of the streaming query, an SPE should either select late merge or global merge. If the query induces high contention on a small set of data values,

e.g., an aggregation with a small group cardinality, local merge is beneficial because it reduces synchronization on the global data structure. In contrast, if a query induces low contention, e.g., an aggregation with a large group cardinality, global merge is beneficial because it omits the additional merging step at the end.

3.3.2.3 Discussion

In this section, we presented different parallelization strategies. UP builds on top of queues to exchange data between operators. This strategy is commonly used in scale-out SPEs in combination with an interpretation-based model. However, as shown in Section 3.2.2, queues are a potential bottleneck for a scale-up optimized SPE. Therefore, we present two late merging strategies that omit queues for data exchange. In particular for a scale-up SPE, those strategies enable a compilation-based execution (see Section 3.3.1). With late merging and a compilation-based query execution, we expect the same improvements for SPEs that Neumann achieved for in-memory databases [24].

3.3.3 Windowing

Windowing is a core feature of streaming systems that splits the conceptually infinite data stream into finite chunks of data, i.e., *windows*. An SPE computes an aggregate for each window, e.g., the revenue per month. As a main goal, an efficient windowing mechanism for an SPE on modern hardware has to minimize the synchronization overhead to enable a massively parallel processing. To this end, we implement a lock-free continuous windowing mechanism following the ideas in [40, 74, 75]. Our implementation uses a double-buffer approach and fine-grained synchronization primitives, i.e., atomic instructions, instead of coarse-grained locks to minimize synchronization overhead. Moreover, our implementation induces a smaller memory footprint than approaches based on aggregate trees [76] as we use on-the-fly aggregation [75] instead of storing aggregate trees.

3.3.3.1 Alternating Window Buffers

In Figure 3.7, we show our double-buffer implementation based on the ideas in [40, 74, 75]. At its core, it exploits alternating window buffers to ensure that there is always one *active buffer* as a destination of incoming tuples. Furthermore, multiple non-active buffers store the results of previous windows. The non-active buffers can be used to complete the aggregate computation, output the results, and reinitialize the buffer memory. As a result, this implementation never defers the processing of the input stream, which increases the throughput of the window operation for both parallelization strategies (see Section 3.3.2). To implement the buffers, we use lock-free data structures that exploit atomic compare-and-swap instructions. This enables concurrent accesses and modification [77, 50, 78]. As a result, several threads can write to a window buffer in parallel and thereby aggregate different parts.

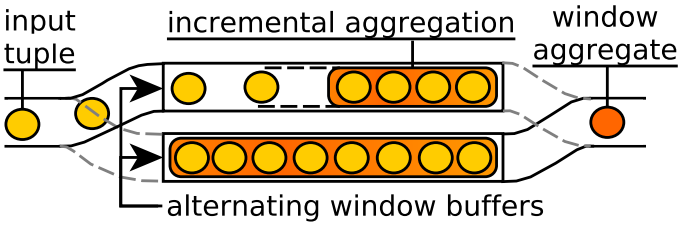


Figure 3.7: Alternating window buffers.

3.3.3.2 Detecting Window Ends

We implement an event-based approach where writer threads check upon the arrival of a tuple if the current window ended. This implies a very low latency for high-bandwidth data streams because arriving tuples cause frequent checks for window ends. Discontinuous streams [79] may lead to high latencies, because of the absence of those checks during the discontinuity. In this case, we switch between our lock-free implementation and a timer-thread implementation depending on the continuity of input streams. A timer thread checks for window ends periodically and, if required, triggers the output. We implement the switch between timer threads and our event-driven technique directly at the source by monitoring the input rate.

3.3.3.3 Extensions

To achieve very low output latencies, we aggregate tuples incrementally whenever possible [75]. To further improve performance, we can combine our implementation with stream slicing [80, 81, 82, 83, 84]. Slicing techniques divide a data stream in non-overlapping chunks of data (*slices*) such that all windows are combinations of slices. When processing sliding windows, we switch alternating buffers upon the start of each new slice. We use time-based windows as an example to simplify the description. However, our windowing technique works with arbitrary stream slicing techniques, which enables diverse window types such as count-based windows [85, 17] and variants of data-driven and user-defined windows [80, 86, 87].

3.3.3.4 Discussion

We implement a lock-free windowing mechanism that minimizes the contention between worker threads. This minimized contention is crucial for a scale-up optimized SPE on modern hardware. To determine its efficiency, we run a micro-benchmark using the data set of the Yahoo! Streaming Benchmark and compare it to a common solution based on timer threads. Our results show that a double-buffer approach achieves about 10% higher throughput on a scale-up SPE on modern hardware.

3.4 Evaluation

In this section, we experimentally evaluate design aspects of an SPE on modern hardware. In Section 3.4.1, we introduce our experimental setup including machine configuration and selected benchmarks. After that, we conduct a series of experiments in Section 3.4.2 to understand the reasons for different throughput values of different implementations and state-of-the-art SPEs. Finally, we summarize and discuss our results in Section 3.4.3.

3.4.1 Experimental Setup

In the following, we present the hardware and software configurations used for our analysis as well as the selected benchmarks and their implementation details.

3.4.1.1 Hardware and Software

We execute our benchmarks on an Intel Core i7-6700K processor with 4 GHz and four physical cores (eight cores using hyper-threading). This processor contains a dedicated 32 KB L1 cache for data and instructions per core. Additionally, each core has a 256 KB L2 cache and all cores share an 8 MB L3 cache. The test system has 32 GB of main memory and runs Ubuntu 16.04. If not stated otherwise, we execute all measurements using all logical cores, i.e., a degree of parallelism of eight.

The C++ implementations are compiled with GCC 5.4 and O3 optimization as well as the *mtune* flags to produce specific code for the underlying CPU. The Java implementations run on the HotSpot VM in version 1.8.0_131. We use Apache Flink 1.3.2, Apache Spark Streaming 2.2.0, and Apache Storm 1.0.0 as scale-out SPEs. We disable fault-tolerance mechanisms (e.g., checkpointing) to minimize the overhead of those frameworks. We use the Streambox (written in C++) release of March 10th and Saber (written in Java) in version 0.0.1 as representative scale-up SPEs. We measure hardware performance counters using Intel VTune Amplifier XE 2017 and the PAPI library 5.5.1.

The JVM-based scale-out SPEs have to serialize and deserialize tuples to send them over the network. The (de)-serialization requires a function call and a memory copy from/to the buffer for each field of the tuple, which adds extra overhead for JVM-based SPEs [57]. In contrast, a scale-up C++ based SPE accesses tuples directly in dense in-memory data structures.

3.4.1.2 Strategies Considered

We implement each benchmark query using the three previously discussed parallelization strategies: upfront partitioning as well as late merging with Local Merge and Global Merge (see Section 3.3.2). The upfront partitioning uses an interpretation-based execution strategy and both Late Merge implementations use a compilation-based execution. Furthermore, all implementations use the lock-free windowing mechanism described in Section 3.3.3. For each strategy and benchmark, we provide a Java and a C++ implementation.

The upfront partitioning strategy follows the design of state-of-the-art scale-out SPEs. Thus, it uses message passing via queues to exchange tuples among operators. In contrast, both Late Merge strategies omit queues and fuse operators where possible. Furthermore, we pass tuples via local variables in CPU registers. In the following, we refer to Late Local Merge and Late Global Merge as Local Merge (LM) and Global Merge (GM), respectively.

In our experiments, we use the memory bandwidth as an upper bound for the input rate. However, future network technologies will increase this boundary [28]. To achieve a data ingestion at memory speed, we ingest streams to our implementations using an in-memory structure. Each in-memory structure is thread-local and contains a pre-generated, disjoint partition of the input. We parallelize query processing by assigning each thread to a different input partition. At runtime, each implementation reads data from an in-memory structure, applies its processing, and outputs its results in an in-memory structure through a sink. Thus, we exclude network I/O and disks from our experiments.

3.4.1.3 Java Optimizations

We optimize our Java implementations and the JVM to overcome well-known performance issues. We ensure that the just-in-time compiler is warmed-up [88] by repeating the benchmark 30 times. We achieve efficient memory utilization by avoiding unnecessary memory copy, object allocations, GC cycles, and pointer chasing operations [89]. Furthermore, we set the heap size to 28 GB to avoid GC overhead, use primitive data types and byte buffer for pure-java implementations and off-heap memory for the scale-out SPEs [90, 91]. Finally, we use the Garbage-First (G1GC) garbage collector, as it performs best for our examined benchmarks [92].

3.4.1.4 Benchmarks

We select the Yahoo! Streaming Benchmark (YSB), the Linear Road Benchmark (LRB), and a query on the New York City Taxi (NYT) dataset to assess the performance of different design aspects for modern SPEs. YSB simulates a real-word advertisement analytics task and its main goal is to assess the performance of windowed aggregation operators [93, 55]. We implement YSB [93] using 10K campaigns based on the codebase provided by [94, 55]. Following these implementations, we omit a dedicated key/value store and perform the join directly in the engine [94, 55].

LRB simulates a highway toll system [95]. It is widely adopted to benchmark diverse systems such as relational databases [95, 96, 97], specialized streaming systems [85, 95, 98, 99], distributed systems [100, 101, 102], and cloud-based systems [34, 103]. We implement the accident detection and the toll calculation for the LRB benchmark [95] as a sub-set of the queries.

The New York City Taxi dataset covers 1.1 billion individual taxi trips in New York from January 2009 through June 2015 [104]. Besides pickup and drop-off locations, it provides additional informations such as the trip time or distance, the fare, or the number of passengers. We divided the area of NY into 1000 distinct regions and formulate the following business relevant

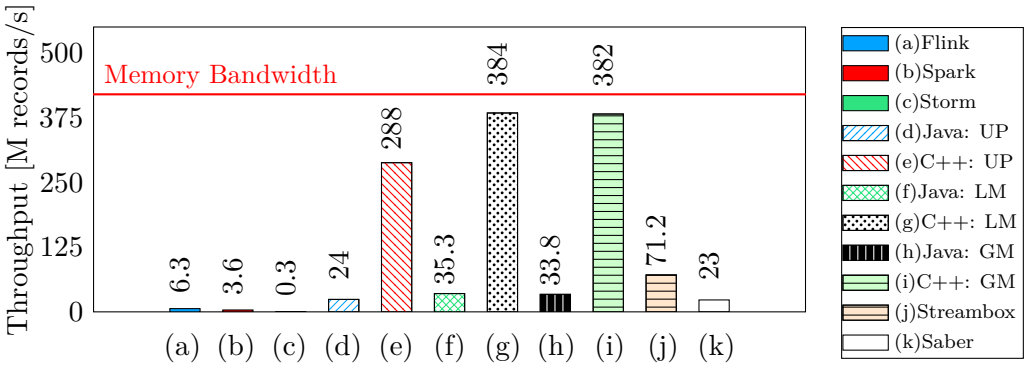


Figure 3.8: YSB single node.

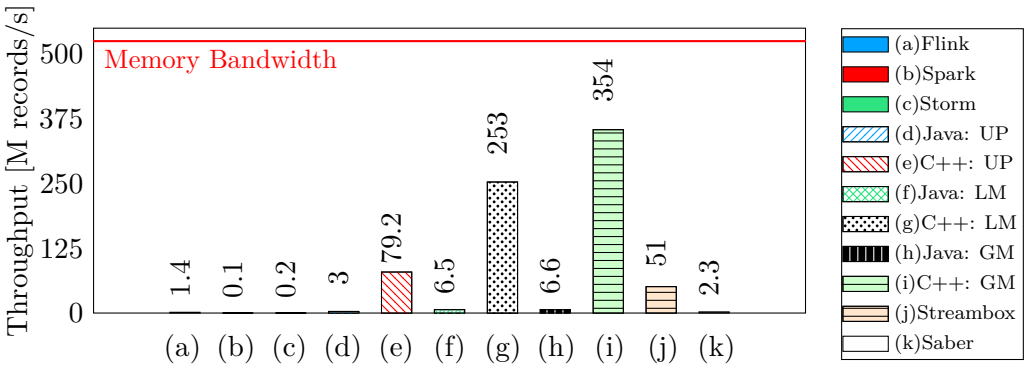


Figure 3.9: LRB single node.

query: *What are the number of trips and their average distance for the VTS vendor per region for rides more than 5 miles over the last two seconds?* The answer to this query would provide a value information for taxi drivers in which region they should cruise to get a profitable ride.

YSB, LRB, and NYT are three representative workloads for todays SPEs. However, we expect the analysis of other benchmarks would obtain similar results. We implement all three benchmarks in Flink, Storm, Spark Streaming, Streambox, and Saber. Note that for SABER, we use the YSB implementation provided in [105] and implement the LRB benchmark and NYT query accordingly. Finally, we provide hand-coded Java and C++ implementations for the three parallelization strategies presented in Section 3.3.2. In our experiments, we exclude the preliminary step of pre-loading data into memory from our measurements.

3.4.1.5 Experiments

In total, we conduct six experiments. First, we evaluate the throughput of the design alternatives showed in this Chapter to get an understanding of their overall performance. In particular, we examine the throughput of the design alternatives for the YSB and LRB (see Section 3.4.2.1). Second, we break down the execution time of the design alternatives to identify the CPU component that consumes the majority of the execution time (see Section 3.4.2.2). Third, we sample data and cache-related performance counters to compare the resource utilization in

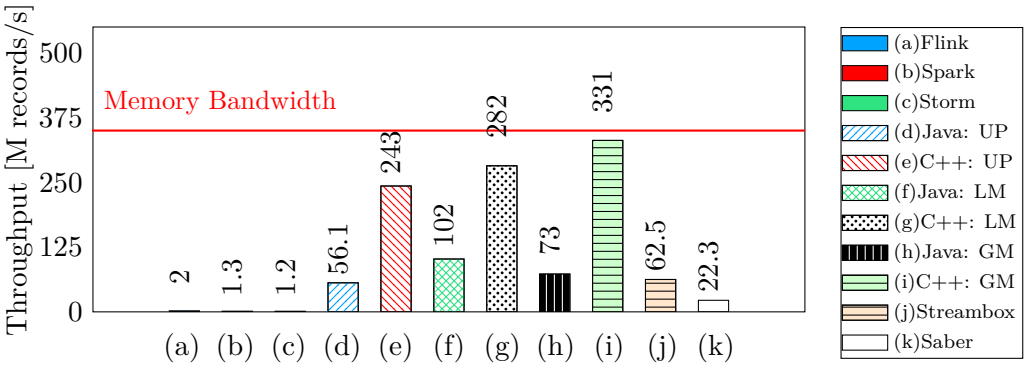


Figure 3.10: NYT query single node.

detail (see Section 3.4.2.3). Fourth, we compare our scale-out optimized implementations of the YSB with published benchmark results of state-of-the-art SPEs running in a cluster (see Section 3.4.2.4). Fifth, we show how scale-out optimizations perform in a distributed execution using a high-speed InfiniBand in Section 3.4.2.5. Finally, we evaluate the impact of design alternatives on latency (see Section 3.4.2.6).

3.4.2 Results

In this section, we present the results of the series of experiments presented in the previous section. The goal is to understand the differences in performance of state-of-the-art SPEs and design alternatives presented in this Chapter.

3.4.2.1 Throughput

To understand the overall efficiency of the design alternatives presented in this Chapter, we measure their throughput in million records per second for both benchmarks in Figures 3.8 and 3.9. Additionally, we relate their throughput to the physically possible memory bandwidth, which could be achieved by only reading data from an in-memory structure without processing it.

YSB Benchmark. The scale-out optimized SPEs Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 1.5% (Flink), 0.8% (Spark Streaming), and 0.07% (Storm). The scale-up optimized Streambox (written in C++) achieves about 17% and Saber (written in Java) achieves 6.6% of the theoretically possible throughput. In contrast, the C++ implementations achieve a throughput of 68.5% (Upfront Partitioning), 91.4% (Local Merge), and 91% (Global Merge) compared to memory bandwidth. For Local and Global Merge, these numbers are a strong indicator that they are bounded by memory bandwidth. We approximate the overhead for using a JVM compared to C++ to a factor of 15 (Upfront Partitioning), 16 (Local Merge), and 13 (Global Merge). We analyze the implementations in-depth in Section 3.4.2.2 and 3.4.2.3 to identify causes of these large differences in performance.

As shown in Figure 3.8, Local and Global Merge outperform the Upfront Partitioning strategy by a factor of 1.34 and 1.33. The main reason for the sub-optimal scaling of Upfront Partitioning is the use of queues. First, queues represent a bottleneck if their maximum throughput is reached (see Section 3.2.2). Second, queues require the compiler to generate two distinct code segments that are executed by different threads such that tuples cannot be passed via registers. Thus, queues lead to less efficient code, which we investigate in detail in Section 3.4.2.3. In contrast, Local and Global Merge perform similar for YSB and almost reach the physical memory limit of a single node. The main reason is that threads work independently from each other because the overall synchronization among threads is minimal.

LRB Benchmark. In Figure 3.9, we present throughput results for the LRB benchmark. Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 0.27% (Flink), 0.03% (Spark Streaming), and 0.01% (Storm). We approximate the overhead of each framework by comparing their throughput with the Java implementation of the upfront partitioning. Such overhead ranges from a factor of 2.7 (Flink), over a factor of 9 (Spark Streaming), to a factor of 39 (Storm). Note that, Spark Streaming processes data streams in micro-batches [36]. We observe that the scheduling overhead for many small micro-batch jobs negatively affects the overall throughput.

The scale-up optimized SPEs Streambox (written in C++) and Saber (written in Java) achieves about 9% and less than 1% of the theoretically possible throughput, respectively. In contrast, the C++ implementations achieve a throughput of 15% (UP), 48% (LM), and 67% (GM) of the physical limit. We approximate the overhead of a JVM compared to C++ to a factor of 20 (UP), 40 (LM), and 55 (GM).

Compared to YSB, the overall throughput of the LRB is lower because it is more complex, requires more synchronization, and larger operator states. In particular, the consumer requires lookups in multiple data structures compared to a single aggregation in the YSB. As a result, concurrent access by different threads to multiple data structures introduces cache thrashing and thus prevents the LRB implementations from achieving a throughput close to the physical limit.

Overall, the Global Merge strategy achieves the highest throughput because it writes in a shared data structure when windows end. In particular, we use a lock-free hash table based on atomic compare-and-swap instructions and open-addressing [77]. Thus, each processing thread writes its results for a range of disjoint keys, which leads to lower contention on the cells of the hash table. As a result, the C++ implementation of the Global Merge outperforms the Java implementation by a factor of 54 because it uses fine-grained atomic instructions (assembly instructions) that are not available in Java. In contrast, the C++ late Merge implementation is slower than Global Merge by a factor of 1.4. The main reason is that the additional merging step at the end of a window introduces extra overhead in terms of thread synchronization. Furthermore, the Java implementation of the Local Merge and Global Merge perform similarly because they implement the same atomic primitives.

The upfront partitioning is slower than Local Merge by a factor of 3.2 because it utilizes queues, buffers, and extra threads, which introduce extra overhead. An analysis of the execution

reveals that the increased code complexity results in a larger code footprint. Furthermore, the upfront partitioning materializes input tuples on both sides of the queue. As a result, the Java implementation of upfront partitioning is slower than the C++ counterpart by a factor of 20.

In the remainder of this evaluation, we use Flink and Streambox as representative baselines for state-of-the-art scale-up and scale-out SPEs. Furthermore, we restrict the in-depth performance analysis to the YSB because the overall results of YSB and LRB are similar.

New York Taxi Query In Figure 3.10, we present the throughput results for the NYT query. As shown, Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 0.5% (Flink), 0.37% (Spark Streaming), and 0.34% (Storm). The pure Java implementations perform faster and exploit 16% (Java UP), 29% (Java LM), and 20% (Java GM) of the available memory bandwidth. In contrast, the C++ based implementation achieve a much higher bandwidth utilization of 69% (C++ UP), 80% (C++ LM), and 94% (C++ GM). Finally, the scale-out optimized SPEs Streambox (17%) and Saber (6%) perform better than the scale-out optimized SPEs. Compared to the YSB, this query induces simpler instructions but each tuple is larger, which leads to better performing pure-Java implementations. However, the the C++ implementations are utilizing the memory bandwidth more efficiently and the best variant (GM) achieves throughput values near the physical memory limit.

3.4.2.2 Execution Time Breakdown

In this section, we break down the execution time for different CPU components following the Intel optimization guide [106]. This time breakdown allows us to identify which parts of the micro-architecture are the bottleneck.

Metrics. Intel provides special counters to monitor buffers that feed micro-ops supplied by the front-end to the out-of-order back-end. Using these counters, we are able to derive which CPU component stalls the CPU pipeline and for how long. In the following, we describe these components in detail. First, the *front-end* delivers up to four micro-ops per cycle to the back-end. If the front-end stalls, the rename/allocate part of the out-of order engine starves and thus execution becomes front-end bound. Second, the *back-end* processes instructions issued by the front-end. If the back-end stalls because all processing resources are occupied, the execution becomes back-end bound. Furthermore, we divide back-end stalls into stalls related to the memory sub-system (called *Memory bound*) and stalls related to the execution units (called *Core bound*). Third, *bad speculation* summarizes the time that the pipeline executes speculative micro-ops that never successfully retire. This time represents the amount of work that is wasted by branch mispredictions. Fourth, *retiring* refers to the number of cycles that are actually used to execute useful instructions. This time represents the amount of useful work done by the CPU.

Experiment. In Figure 3.11, we breakdown the execution of cycles spent in the aforementioned five CPU components based on the Intel optimization guide [106].

Flink is significantly bounded by the front-end and bad speculation (up to 37%). Furthermore, it spends only a small portion of its cycles waiting on data (23%). This indicates that Flink is

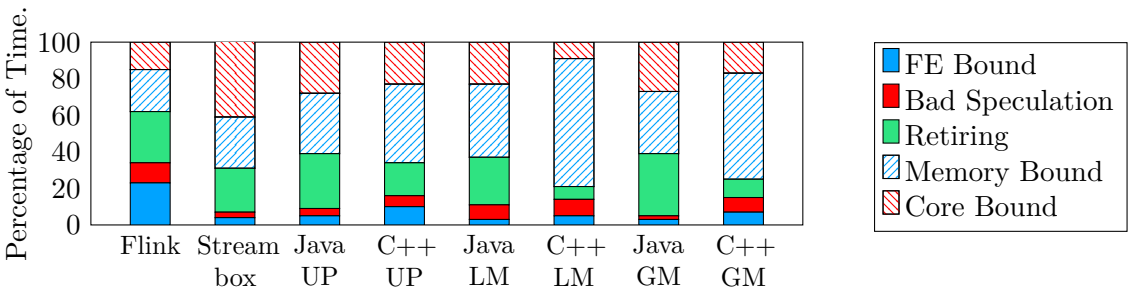


Figure 3.11: Execution time breakdown YSB.

CPU bound. In contrast, Streambox is largely core bounded which indicates that the generated code does not use the available CPU resources efficiently.

All Java implementations are less front-end bound and induce less bad speculation and more retiring instructions compared to their respective C++ implementations. In particular, the Java implementations spend only few cycles in the front-end or due to bad speculation (5%-11%). The remaining time is spent in the back-end (23%-28%) by waiting for functional units of the out-of-order engine. The main reason for that is that Java code accesses tuples using a random access pattern and thus spend the majority of time waiting for data. Therefore, an instruction related stall time is mainly overlapped by long lasting data transfers. In contrast, C++ implementations supply data much faster to the CPU such that the front-end related stall time becomes more predominant. We conclude that Java implementations are significantly *core bound*. LRB induces similar cycle distributions and is therefore not shown.

All C++ implementations are more memory bound (43%-70%) compared to their respective Java implementations (33%-40%). To further investigate the memory component, we analyze the exploited memory bandwidth. Flink and Streambox utilize only 20% of the available memory bandwidth. In contrast, C++ implementations utilize up to 90% of the memory bandwidth and Java implementations up to 65%. In particular, the Java implementations exploit roughly 70% of the memory bandwidth of their respective C++ implementations. Therefore, other bottlenecks, such as inefficient resource utilization inside the CPU, may prevent a higher record throughput. Finally, the more complex code of LRB results in 20% lower memory bandwidth among all implementations (not shown here). We conclude that C++ implementations are *memory bound*.

Outlook. The memory bound C++ implementations can benefit from an increased bandwidth offered by future network and memory technologies. In contrast, Java implementations would benefit less from this trend because they are more core bound compared to the C++ implementations. In particular, a CPU becomes core bound if its computing resources are inefficiently utilized by resource starvation or non-optimal execution ports utilization. In the same way, state-of-the-art scale-out SPEs would benefit less from increased bandwidth because they are significantly front-end bound. In contrast, Streambox as a scale-up SPE would benefit from an improved code generation that utilizes the available resources more efficiently. Overall, Java and current SPEs would benefit from additional resources inside the CPU, e.g., register or compute units, and from code that utilizes the existing resources more efficiently.

	Flink	Stream-box	Java: UP	C++: UP	Java: LM	C++: LM	Java: GM	C++: GM
Branches /Input Tuple	549	350	418	6.7	191	2.7	192	2.6
Branch Mispred./ Input Tuple	1.84	2.53	1.44	0.37	0.77	0.37	0.79	0.36
L1D Misses/ Input Tuple	52.16	42.4	12.86	1.93	5.66	1.41	7.31	1.42
L2D Misses/ Input Tuple	81.93	107.09	34.42	3.54	14.50	2.22	23.79	3.66
L3 Misses/ Input Tuple	37.26	72.04	13.97	1.67	6.88	1.56	6.70	1.83
TLBD Misses/ Input Tuple	0.225	0.825	0.145	0.00035	0.01085	0.000175	0.00292	0.0012
L1I Misses/ Input Tuple	19.5	1.35	0.125	0.00125	0.0155	0.0002	0.0229	0.000075
L2I Misses/ Input Tuple	2.2	0.6	0.075	0.001	0.008	0.0001	0.009	0.00007
TLBI Misses/ Input Tuple	0.04	0.3	0.003	0.0004	0.0007	0.000008	0.0009	0.000003
Instr. Exec./ Input Tuple	2,247	2,427	1,344	136	699	49	390	64

Table 3.1: Resource utilization of design alternatives.

3.4.2.3 Analysis of Resource Utilization

In this section, we analyze different implementations of YSB regarding their resource utilization. In Table 3.1, we present sampling results for Flink and Streambox as well as Upfront Partitioning (Part.), Local Merge (LM), and Global Merge (GM) implementations in Java and C++. Each implementation processes 10M records per thread and the sampling results reflect the pure execution of the workload without any preliminary setup.

Control Flow. The first block of results presents the number of branches and branch mispredictions. These counters are used to evaluate the control flow of an implementation. As shown, the C++ implementations induce only few branches as well as branch mispredictions compared to the Java implementations (up to a factor of 3) and Flink (up to a factor of 5). However, all C++ implementations induce the same number of branch mispredictions but different number of branches. The main reason for that is that UP induces additional branches for looping over the buffer. Since loops induce only few mispredictions, the number of mispredictions does not increase significantly.

Overall, C++ implementations induce a lower branch misprediction rate that wastes fewer cycles for executing mispredicted instructions and loads less unused data. Furthermore, a low branch misprediction rate increases code locality. Regarding the state-of-the-art SPEs, Streambox induces less branches but more branch mispredictions.

Data Locality. The second block presents data cache related counters which can be used to evaluate the data locality of an implementation. Note that, YSB induces no tuple reuse because a tuple is processed once and is never reloaded. Therefore, data-related cache misses are rather high for all implementations. As shown, the C++ implementations outperform the Java based implementations by inducing the least amount of data cache misses per input tuple in all three cache levels. The main reason is the random memory access pattern that is introduced by object scattering in memory for JVM-based implementations. Furthermore, all C++ implementations

produce similar numbers but the Java implementations differ significantly. Interestingly, the Java Upfront Partitioning implementation and Flink induce a significantly higher number of misses in the data TLB cache, which correlates directly to their low throughput (see Figure 3.8). Additionally, Streambox induces the most data related cache and TLB misses which indicates that they utilize a sub-optimal data layout.

Overall, C++ implementations induce up to 9 times less cache misses per input tuple which leads to a higher data locality. This higher data locality results in shorter access latencies for tuples and thus speeds up execution significantly.

Code Locality. The third block presents instruction cache related counters, which can be used to evaluate the code locality of an implementation. The numbers show that the C++ implementations create more efficient code that exhibits a high instruction locality with just a few instruction cache misses. The high instruction locality originates from a small instruction footprint and only few mispredicted branches. This indicates that the instructions footprint of the C++ implementations of YSB fits into the instruction cache of a modern CPU. Furthermore, the late merging strategies produce significantly fewer cache misses compared to the Upfront Partitioning strategy. The misses in the instruction TLB follow this behavior.

The last block presents the number of executed instructions per input tuple which also impacts the code locality. Again, the C++ implementations execute significantly fewer instructions compared to Java implementations. Furthermore, the late merging strategies execute fewer instructions than the UP. This is also reflected in the throughput numbers in Figure 3.8. Since the C++ implementations induce fewer instruction cache misses compared to the respective Java implementations, we conclude a larger code footprint and the virtual function calls of a JVM are responsible for that. Interestingly, Flink executes almost as many instructions as Streambox but achieves a much lower throughput.

For all instruction related cache counters, Flink induces significantly more instruction cache misses compared to Java implementations by up to three orders of magnitude. The main reason for this is that the code footprint of the generated UDF code exceeds the size of the instruction cache. In contrast, Streambox generates smaller code footprints and thus induces less instruction related cache misses.

Summary. All C++ implementations induce a better control flow, as well as a higher data and instruction locality. In contrast, Java-based implementations suffer from JVM overheads such as pointer chasing and transparent memory management. Overall, both late merging strategies outperform Upfront Partitioning. Finally, Flink and Streambox are not able to exploit modern CPUs efficiently.

3.4.2.4 Comparison to Cluster Execution

In this section, we compare our hand-written implementations with scale-out optimized SPEs on a cluster of computing nodes. Additionally, we present published results from the scale-up SPEs SABER and Streambox. In Figure 3.12, we show published benchmarking results of YSB on a

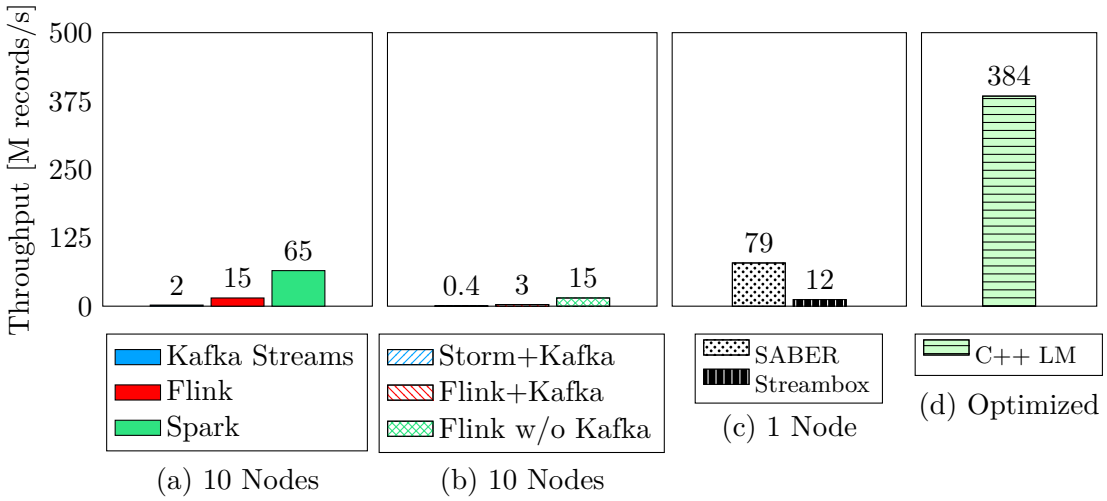


Figure 3.12: YSB reported throughput.

cluster [94, 107, 108]. We choose to use external numbers because the vendors carefully tuned their systems to run the benchmark as efficiently as possible on their cluster.

The first three numbers are reported by databricks [107] on a cluster of 10 nodes. They show that *Structured Streaming*, an upcoming version of Apache Spark, significantly improves the throughput (65M rec/sec) compared to Kafka Streams (2M rec/sec) and Flink (15M rec/sec). The next three numbers are reported by dataArtisans [94] on a cluster of 10 nodes. They show that Storm with Kafka achieves only 400K recs/sec and Flink using Kafka achieves 3M rec/second. However, if they omit Kafka as the main bottleneck and move data generation directly into Flink, they increase throughput up to 15M recs/sec.

Figure 3.12 shows, even when using 10 compute nodes, the throughput is significantly below the best performing single node execution that we present in this Chapter (C++ LM). The major reason for this inefficient scale-out is the partitioning step involved in distributed processing. As the number of nodes increases, the network traffic also increases. Therefore, the network bandwidth becomes the bottleneck and limits the overall throughput of the distributed execution. However, even without the limitations on network bandwidth by using one node and main memory, state-of-the-art SPEs cannot utilize modern CPUs efficiently (see Figure 3.8). One major reason for this is that their optimization decisions are focused on scale-out instead of scale-up.

The next two benchmarking results are reported in [105] which measure the throughput of SABER and Streambox on one node. Although they achieve a higher throughput compared to scale-out SPEs, they still do not exploit the capabilities of modern CPUs entirely. Note that the numbers reported in [105] differ from our numbers because we use different hardware. Furthermore, we change the implementation of YSB on Streambox to process characters instead of strings and optimize the parameter configuration such that we can improve the throughput to 71M rec/sec (see Figure 3.8). The last benchmarking result reports the best performing hand-coded C++ implementation presented in this Chapter which achieves a throughput of up to 384M rec/sec.

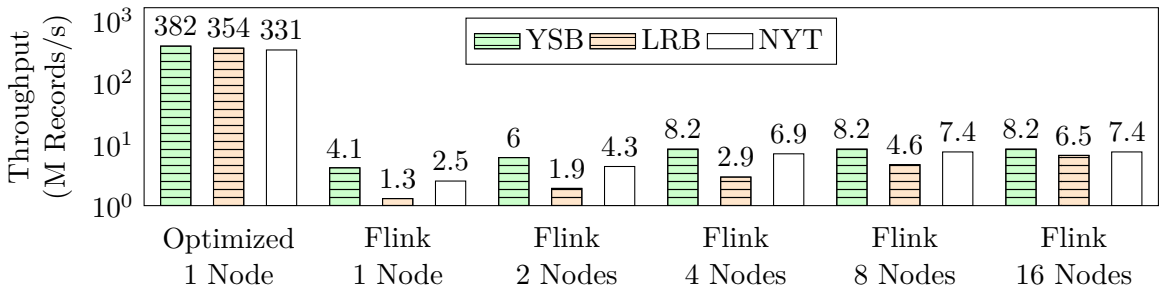


Figure 3.13: Scale-out experiment using YSB, LRB, and NYT query.

As a second experiment, we demonstrate the scale-out performance of Flink on the YSB, LRB, and NYT and compare it to a single node implementation. Note that, we did the same scale-out experiments for all queries on Spark and Storm and they show the same behavior; therefore, we omit them in Figure 3.13. We execute the benchmarks on up to 16 nodes, whereas each node contains a 16-core Intel Xeon CPU (E5620 2.40GHz) and all nodes are connected via 1Gb Ethernet. In Figure 3.13, we compare our single node optimized version (DOP=1) with the the scale-out execution (DOP 2-16). As shown, YSB scales to up four nodes with a maximum throughput of 8.2M recs/sec. In contrast, NYT scales to up eight nodes with a maximum throughput of 7.4M recs/sec. The main reason for this sub-optimal scaling is that the benchmarks are network-bound. Flink generates the input data inside the engine and uses its state management APIs to save the intermediate state. However, the induced partitioning utilizes the available network bandwidth entirely. In contrast, LRB does increase the throughput for each additional node but the overall increase is only minor.

Overall, Figure 3.13 reveals that even if we scale-out to a large cluster of nodes, we are not able to provide the same throughput as a hand-tuned single node solution. However, a recent study shows that as the network bandwidth increases, the data processing for distributed engines becomes CPU-bound [57]. The authors conclude that current data processing systems, including Apache Spark and Apache Flink need systematic changes.

3.4.2.5 Remote Direct Memory Access

In this experiment, we want to examine if hand-written Java or C++ implementations are capable of utilizing the ever rising available network bandwidth in the future. To this end, we extend the YSB Benchmark such that one node produces the data, as it will be the case in the streaming scenario, and one node gets the stream of data via Infiniband and RDMA. In Figure 3.14, we show the throughput of our hand-written implementation of the YSB using Java and C++ on an E7-4850v4 CPU and an InfiniBand FDR network. With *physical limit*, we refer to the maximum bandwidth following the specification of 56 Gbit/s (FDR 4x). For Java as well as for C++, we implement one version that just sends data via the network without processing it, i.e., read only, and one version that includes the processing of the YSB, i.e., with proc. As shown, a C++ implementation that sends data over a high-speed InfiniBand network and processes it is able to

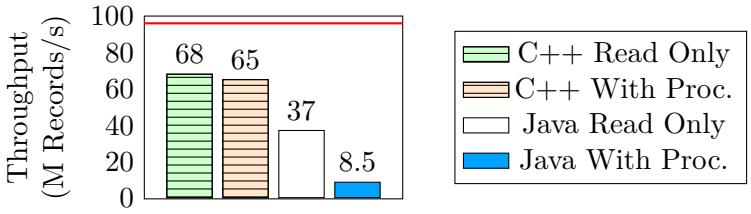


Figure 3.14: Infiniband exploitation using RDMA (the red line indicates the physical limit).

reach nearly the same throughput as the implementation without processing. In contrast, the Java implementation with processing achieves only 22% of the throughput without processing. This indicates that the Java implementation is bounded by the processing instead of the network transfer. This is in line with the results presented in the previous sections. Overall, the C++ implementation is able to exploit 70% of the theoretical bandwidth without processing compared to 38% using Java.

3.4.2.6 Latency

In this experiment, we compare the latencies of the Upfront Partitioning and the Global Merge parallelization strategy for a C++ implementations processing and 10M YSB input tuples. With latency, we refer to the time span between the end of a window and the materialization of the window at the sink. Our results show, that the Global Merge exhibits an average latency of 57ns (min: 38ns; max: 149ns). In contrast, Upfront Partitioning induces a one order of magnitude higher latency of 116μs (min: 112μs; max: 122μs). The main reason for this is the delay introduced by queues and buffers.

In all our experiments, Global Merge implementations achieve very low latencies in the order of a hundred nanoseconds. This is in contrast to state-of-the-art streaming systems which use Upfront Partitioning [20, 22, 2]. In particular, these systems have higher latencies (in the order of ms) for the YSB [55, 94, 108]. The Linear Road Benchmark defines a fixed latency upper bound of 5 seconds which our implementation satisfies by far.

3.4.3 Discussion

Our work is driven by the key observation that emerging network technologies allow us to ingest data with main memory bandwidth on a single scale-up server [28]. We investigate design aspects of an SPE optimized for scale-up. The results of our extensive experimental analysis are the following key insights:

Avoid high level languages. Implementing the critical code path or the performance critical data structures in Java results in a larger portion of random memory accesses and virtual function calls. These issues slow down Java implementations up by a factor of 54 compared to our C++ implementation. Furthermore, our profiling results provide strong indication that current SPEs are CPU-bound.

Avoid queuing and apply operator fusion. An SPE using queues as a mean to exchange data between operators is not able to fully exploit the memory bandwidth. Thus, data exchange should be replaced by operator fusion, which is enabled by query compilation techniques [24].

Use Late Merge parallelization. The parallelization based on Upfront Partitioning introduces an overhead by up to a factor of 4.5 compared to late merging that is based on logically dividing the stream in blocks. The key factor that makes working on blocks applicable in an SPE is that the network layer performs batching on a physical level already.

Use lock-free windowing. Our experiments reveal that a lock-free windowing implementation achieves a high throughput for a SPE on modern hardware. Furthermore, our approach enables queue-less late merging.

Scale-Up is an alternative. In our experiments, a single node using an optimized C++ implementation outperforms a 10 node cluster running state-of-the-art streaming systems. We conclude that scale-up is a viable way of achieving high throughput and low latency stream processing.

3.5 Related Work

In this section, we cover additional related work that we did not discuss already. We group related work by topics.

3.5.1 Stream Processing Engines

The first generation of streaming systems has built the foundation for today's state-of-the-art SPEs [85, 109, 110, 111]. While the first generation of SPEs explores the design space of stream processing in general, state-of-the-art systems focus on throughput and latency optimizations for high velocity data streams [14, 112, 20, 34, 21, 22, 113, 2]. Apache Flink [20], Apache Spark [2], and Apache Storm [22] are the most popular and mature open-source SPEs. These SPEs optimize the execution to *scaling-out* on shared-nothing architectures. In contrast, another line of research focuses on *scaling-up* the execution on a single machine. Their goal is to exploit the capabilities of one high-end machine efficiently. Recently proposed SPEs in this category are Streambox [4], Trill [39], or Saber [40]. In this Chapter, we examine the data-related and processing-related design space of scale-up and scale-out SPEs in regards to the exploitation of modern hardware. We showcase design changes that are applicable in many SPEs to achieve higher throughput on current and future hardware technologies. In particular, we lay the foundation by providing building blocks for a third generation SPE, which runs on modern hardware efficiently.

3.5.2 Data Processing on Modern Hardware

In-memory databases explore fast data processing near memory bandwidth speed [114, 115, 24, 116, 117]. Although in-memory databases support fast state access [118, 119], their overall stream processing capabilities are limited. In this Chapter, we widen the scope of modern hardware

exploitation to more complex stream processing applications such as LRB, YSB, and NYT. In this field, SABER also processes complex streaming workloads on modern hardware [40]. However, it combines a Java-based SPE with GPGPU acceleration. In contrast, we focus on outlining the disadvantages of a Java-based SPEs. Nevertheless, the design changes proposed by the authors of SABER are partially applicable for JVM-based SPEs.

3.5.3 Performance Analysis of SPEs

Zhang et al. [16] deeply analyze the performance characteristics of Storm and Flink. They contribute a NUMA-aware scheduler as well as a system-agnostic not-blocking tuple buffering technique. Complementary to the work of Zhang, we investigate fundamental design alternatives for SPEs, specifically data passing, processing models, parallelization strategies, and an efficient windowing technique. Furthermore, we consider upcoming networking technologies and existing scale-up systems.

Ousterhout et al. [120] analyze Spark and reveal that many workloads are CPU-bound and not disk-bound. In contrast, our work focuses on understanding why existing engines are CPU-bound and on systematically exploring means to overcome these bottlenecks by applying scale-up optimizations.

Trivedi et al. assess the performance-wise importance of the network for modern distributed data processing systems such as Spark and Flink [57]. They analyze query runtime and profile the systems regarding the time spent in each major function call. In contrast, we perform a more fine-grained analysis of hardware usage of those and other SPEs and native implementations.

3.5.4 Query Processing and Compilation

In this Chapter, we used hand-coded queries to show potential performance advantages for SPEs using query compilation. Using established code generation techniques proposed by Krikellas [121] or Neumann [24] allows us to build systems that generate optimized code for any query, i.e., also streaming queries. However, as opposed to a batch-style system, input tuples are not immediately available in streaming systems. Furthermore, the unique requirements of windowing semantics introduce new challenges for a query compiler. Similarly to Spark, we adopt code generation to increase throughput. In contrast, we directly generate efficient machine code instead of byte code, which requires an additional interpretation step for the execution.

3.5.5 Stream Processing on Modern Hardware

BriskStream [6] is a JVM-based SPE that efficiently executes streaming queries on multi-core, multi-socket CPUs. To this end, BriskStream schedules operators on the CPU cores to reduce intra-socket communication. Our techniques are orthogonal to BriskStream, as we focus on increasing the efficiency of stream processing on modern hardware via a fundamental redesign of the architecture of an SPE. System builders who seek to accelerate their SPEs on multi-

socket architecture benefit from our techniques as well as the solution proposed by the authors of BriskStream.

The solutions listed below apply query compilation to speed up stream processing workloads following the ideas proposed in this Chapter. In particular, Grizzly [38] and LightSaber [5] are SPEs prototypes propose scale-up execution runtimes that rely on query compilation to achieve operator fusion. Grizzly uses adaptive query compilation to continuously re-optimize the generated code of running queries to adjust it in the changes in data characteristics. In contrast, LightSaber offers query compilation along with a fine-grained late merging execution model based on parallel aggregation trees. Overall, the two SPEs address the limitations of JVM-based SPEs by following the guidelines discussed in this Chapter.

3.6 Conclusion

In this Chapter, we have analyze the design space alternatives to build SPEs optimized for scale-up. Our experimental analysis reveals that the design of current SPEs cannot exploit the full memory bandwidth and computational power of modern hardware. In particular, SPEs written in Java cause a larger number of random memory accesses and virtual function calls compared to C++ implementations.

Our hand-written implementations that use appropriate streaming optimizations for modern hardware achieve near memory bandwidth and outperform existing SPEs in terms of throughput by up to two orders of magnitude on three common streaming benchmarks. Furthermore, we show that carefully-tuned single node implementations outperform existing SPEs even if they run on a cluster. Emerging network technologies deliver similar or even higher bandwidth compared to main memory. Current systems cannot fully utilize the current memory bandwidth and, thus, also cannot exploit faster networks in the future.

With this Chapter, we lay the foundation for a scale-up SPE by exploring design alternatives in-depth and describing a set of design changes that can be incorporated into current and future SPEs. In particular, we show that a queue-less execution engine based on query compilation, which eliminates queues as a tool to exchange intermediate results between operators, is highly suitable for an SPE on modern hardware. Furthermore, we suggest to replace Upfront Partitioning with Late Merging strategies.

Overall, we conclude that with emerging hardware, scale-up is a viable alternative to scale-out, if the resource of a single-node instance can efficiently sustain the workload. In the next Chapter, we make use of the findings of this Chapter and investigate how to accelerate those workloads that do not fit the single-node instance, i.e., by efficiently scaling-out streaming query execution on a cluster with high-speed networks.

4

Rethinking Stateful Stream Processing on High-speed Networks

4.1 Introduction

In the previous Chapter, we provide solutions to rethink the single-node SPE design to efficiently exploit modern hardware query processing performance for stateful stream processing workloads. In this Chapter, we reuse the concepts of the previous Chapter to design an SPE that efficiently executes stateful stream processing workloads on distributed computing systems. We observe that the advancement in data center networking technology, over the last decade, has bridged the gap between network and main memory data rates [23, 28]. In fact, it is possible today to purchase or rent servers that offer supercomputer-grade network bandwidths [122, 123]. For instance, a high-speed Network Interface Controller (NIC) supports up to 25 GB/s as network throughput and 600 ns latency per port [53], while modern switches support up to 40 TB/s as overall network throughput [124]. Double Data Rate 4 (DDR4) modules support up to 25 GB/s per channel and 13 ns CAS latency, while main memory bandwidth reaches up to 204.8 GB/s [125]. This improvement is due to Remote Direct Memory Access (RDMA): a feature of high-speed networks that enables high throughput data transfer with microsecond latency. Thus, the common assumption that network is often the bottleneck in distributed settings no longer holds [29].

Research has shown that RDMA hardware does not provide a "plug-and-play" performance gain to existing data management systems [28]. Consequently, it is necessary to revise their architecture to use full bandwidth [28]. Recent research has proposed a number of architecture revisions to accelerate OLAP [28, 126], OLTP [127], indexing [128], and key-value stores [129, 54] using RDMA in rack-scale deployments. In this Chapter, we make the case that Stream Processing Engines (SPEs) also require architectural changes to truly benefit from RDMA

hardware. To this end, we show that current scale-out SPEs are not ready for RDMA acceleration and existing RDMA solutions do not fit the stream processing paradigm. Thus, we propose an SPE architecture that natively integrates with RDMA to efficiently ingest and process data in rack-scale deployments.

Current SPEs, e.g., Apache Flink [35], Storm [22], TimelyDataflow [21], and Spark [36], cannot fully benefit from RDMA hardware. Their design choices fundamentally prevent them from processing data at full data-center network speed for the following reasons. First, *RDMA-unfriendliness,* current SPEs rely on socket-based networking, e.g., TCP/IP, to ingest and exchange data streams. Even though socket-based networking runs on RDMA hardware, it cannot fully exploit its potential, e.g., using IP-over-InfiniBand (IPoIB) [28]. Second, *costly message-passing*, current SPEs rely on message-passing to process data following a Map/Reduce-like paradigm [49]. This results in a performance regression and thus in an inefficient execution induced by sub-optimal data and code locality [47, 16]. In particular, message passing induces expensive queue-based synchronization among network and data processing threads [27]. Furthermore, Map/Reduce-like paradigms are network-bound on relatively slow socket-based connections, when considering data-intensive workloads. Yet, they become compute bound in the presence of fast networks [28, 29]. As a result, they do not benefit from the data rate of a high-speed network. Finally, *costly scale-out execution*, current scale-out SPEs rely on *operator fission* [46] to achieve data-parallel computations. This enables each SPE executor to process a disjoint partition of the stream and manage local state. However, fission involves continuous data re-partitioning, which is expensive [47].

Furthermore, previous RDMA solutions for data-intensive systems do not solve the above problems. An SPE requires to run stateful analytics on in-flight records and perform point updates and range scans on operator state. However, RDMA-accelerated OLAP systems speed-up batch analytics on immutable datasets [130, 58, 131, 126]. RDMA-based key-value stores are design for transactional workloads comprising point lookups and insertions [54, 129]. As a result, the data-access patterns and processing model of SPEs need dedicated solutions for RDMA-acceleration.

To enable stateful stream processing at full network bandwidth with very low latency, we propose Slash, our RDMA-accelerated SPE for rack-scale deployments. We design a new architecture to enable a processing model that omits data re-partitioning and applies stateful query logic on ingested streams. Slash comprises the following building blocks: the RDMA Channel, the stateful query executor, and the Slash State Backend (SSB). First, we design an RDMA-friendly protocol to support streaming among nodes via dedicated RDMA data channels. This enables Slash to perform data ingestion and data exchange among nodes at full RDMA network speed, by leveraging the aggregated bandwidth of all NICs. Second, we devise a stateful executor that omits message passing and runs queries following late merge technique [47]. Finally, we replace re-partitioning with the SSB that enables consistent state sharing across distributed nodes. This enables multiple nodes to concurrently update the same key-value pair of the state

(for instance, a group of a windowed aggregation). To ensure consistency, we introduce an epoch-based protocol to lazily synchronize state updates using RDMA.

Slash executors scale out computation by eagerly applying stateful operators on data stream to compute partial state. Executors store partial state into the SSB, which ensures a consistent view of the state for all Slash executors. Our evaluation on common streaming workloads shows that Slash outperforms baseline approaches based on data re-partitioning and is skew-agnostic. In particular, we compare Slash against a scale-out SPE (Apache Flink) on an IPoIB network, a scale-up SPE called LightSaber, and a self-developed straw-man solution called RDMA UpPar, which scales out query execution via RDMA-based data re-partitioning. Slash achieves up to 22x and 11.6x higher throughput than RDMA UpPar and LightSaber, respectively. Furthermore, Slash outperforms Flink by achieving an order of magnitude higher throughput. In sum, this shows that RDMA alone cannot achieve peak performance without redesigning the SPE internals.

In this Chapter, we make the following contributions.

- To natively integrate high-speed RDMA networks, we propose Slash, a novel RDMA accelerated SPE.
- We design a stateful query executor to make Slash scale-out data stream processing over an RDMA network.
- We define an RDMA streaming protocol for Slash to transfer data at line rate using the RDMA channel.
- We architect the SSB that enables a distributed consistent state over RDMA interconnects.
- We validate Slash's design on common streaming benchmarks on a high-end RDMA cluster and show up to 25x throughput improvement over our strongest baseline.

We structure this Chapter as follows. In Section 4.2, we make the case of RDMA acceleration for an SPE and list challenges and opportunities. In Section 4.3, we present the system architecture of Slash and provide an overview of each component. After that, we describe the stateful executor (see Section 4.4), the RDMA Channel (see Section 4.5), and the SSB (see Section 4.6). Afterwards, we conduct an extensive evaluation of Slash in Section 4.7. We describe related works in the realm of SPEs and RDMA-enabled database systems in Section 4.8. Finally, we summarize the findings of this Chapter and discuss ideas for future work in Section 4.9.

4.2 The Case for RDMA-Accelerated Stateful Stream Processing

In this section, we make the case for RDMA-based acceleration of stateful stream processing workloads. To this end, we analyze options to co-design an SPE with RDMA networks (see Section 4.2.1) and derive design challenges to be tackled (see Section 4.2.2). Afterwards, driven by our analysis, we propose the system architecture of Slash (see Section 4.3).

4.2.1 RDMA Integration

Previous research proposes three general approaches for the integration of RDMA into a general-purpose data management system [28]. In the following, we analyze their applicability to an SPE and the implication behind these design decisions.

Plug-and-play integration. In this approach, the deployment of a shared-nothing system occurs on an IPoIB network. Previous research has shown that it does not necessarily result in performance improvements [28]. In particular, IPoIB does not saturate network bandwidth and introduces CPU overhead for small messages [28]. In our evaluation, we show that query execution of current SPEs improves only slightly using IPoIB.

Lightweight integration. This approach involves the replacement of socket-based networking with RDMA verbs. Although it benefits from high network bandwidth, it still suffers from bottlenecks that are present in the original design [28, 52, 47]. We validate this claim by building a straw-man solution that implements the lightweight approach. In particular, we implement and evaluate a data re-partitioning component that uses RDMA QPs instead of sockets [126]. Note that several SPEs, such as Apache Flink and TimelyDataflow, leverage data re-partitioning to parallelize operators and thus scale-out computation. We refer to this approach in the remainder of this Chapter as *RDMA UpPar*. Note that the performance regression induced by lightweight integration does not depend on the underlying runtime or language. Thus, we implement RDMA UpPar in C++ to omit managed runtime overhead and show that the performance regression is due to the overall SPE design.

Native integration. Previous research indicates that the most-effective option for the integration is to co-design software components with RDMA [28, 52, 128, 127]. For a data management system, this approach involves a re-design of its internals, e.g., storage management and query processing, to remove the network bottleneck. Notably, the data re-partitioning component of a scale-out SPE is network bound in the presence of a high data rates and slow networks [29]. Furthermore, recent works have shown that data re-partitioning is responsible for performance regressions in scale-out SPEs due to its sub-optimal CPU utilization in the presence of high bandwidths [16, 47]. This suggests that an SPE must re-think its scale-out computational model and evaluate alternatives to data re-partitioning, to fully benefit from RDMA acceleration. We refer to this approach in the remainder of this Chapter as *Slash*.

With Slash, we focus on the native approach and discuss the necessary architectural and algorithmic changes when designing an RDMA-accelerated SPE. In addition, we assess the performance of the native solution against plug-and-play and lightweight integrations.

4.2.2 Design Challenges

In the previous section, we make the case for native RDMA integration, which promises peak performance, but requires changes to a scale-out SPE. In this section, we analyze the design challenges and the implications that come with native RDMA acceleration.

C1: Efficient Streaming Computations. Besides high-throughput data transfer, RDMA-based systems enable a highly-efficient decoupled processing model: storage and compute nodes access each other's memory at byte-level granularity [28, 126, 127, 129, 128]. In this model, costly data re-partitioning for data-parallel processing is not necessary, as each node can access data from any remote memory location. As a result, we identify as first design challenge to support an efficient processing model for distributed streaming computations that profits from memory access at byte-level granularity. This involves replacing re-partitioning strategy with a performance-friendly, RDMA-based processing strategy.

C2: Efficient Data Transfer. In an SPE, data channels must enable efficient data transfer among operators over the network. However, RDMA provides several network transfer capabilities, which induce many design options and trade-offs between throughput and latency. For instance, one-sided verbs achieve lower network latency than two-sided verbs, especially on the most recent NICs [132, 27, 54]. An RDMA READ involves a full network round-trip and has thus higher latency than an RDMA WRITE [52]. Furthermore, techniques, such as huge-pages, pipelining, header-only messages, and selective signaling, further improve performance [52, 28]. To the best of our knowledge, there is no comprehensive analysis of RDMA capabilities on stream processing workloads [47, 133]. As a result, we identify as second design challenge the selection of RDMA capabilities to achieve high-throughput stream processing with low latency.

C3: Consistent Stateful Computation. SPEs needs to ensure consistent stateful computation. Thus, an RDMA-accelerated SPE must consistently manage state, while processing incoming records. This involves keeping track of the distributed computation, while ensuring exactly-once state updates. As a result, we identify as third design challenge to achieve consistency guarantees for stateful streaming operators that access shared data structures using RDMA.

In sum, there is no system that fully solves the above design challenges, to enable stateful stream processing at RDMA speed. Therefore, we propose Slash in the next section as a solution to bridge the gap between RDMA acceleration and stream processing.

4.3 System design

The architecture of Slash comprises three components to enable robust stream processing: the *stateful executor* (1), the *RDMA channel* (2), and the *Slash State Backend* (3). Figure 4.1 shows that each node executes an instance (a process) of a Slash stateful executor, which reads and writes stream records using RDMA channels, applies operator logic, and stores intermediate state into the state backend.

Stateful executor. A guiding design principle for the stateful executor of Slash (see Section 4.4) is to make the common case fast when leveraging RDMA acceleration. To this end, Slash executors follow a relaxed processing model based on lazy merging of eagerly computed partial states. This is similar to the execution model of scale-up SPEs, which is based on late merge. However, Slash performs merging at cluster level using RDMA. To avoid costly data repartitioning, Slash executor eagerly compute partial state in parallel on physical data

flows of a stream Furthermore, Slash executors lazily merge partial, distributed state and output consistent result using our RDMA-based components: the SSB and the RDMA channels. We use RDMA acceleration to design efficient distributed algorithms and data structures that enable fast, coherent memory access at byte-level granularity.

RDMA Channel. Slash RDMA channels (see Section 4.5) are data channels that enable sending and receiving records at full line rate with sub-millisecond latency, via an RDMA-shared circular queue. Sender and receiver read/write from/to the queue using RDMA semantics. In contrast to socket-based RDMA channels, our RDMA channels enable higher throughput transfer and zero-copy semantics. We use RDMA channels to implement data repartitioning in RDMA UpPar as well as communication primitive for Slash.

Slash State Backend. The SSB is a distributed state backend (see Section 4.6) that maintains operator state across the aggregated memory of multiple nodes. We design our distributed state backend for common stream processing use-cases: update-intensive workloads, and quick triggering and post-processing of in-flight windows [134]. In contrast to traditional approaches, which allow for local mutable state co-partitioned with input stream, our state backend enables distributed mutable state using RDMA. As a result, Slash executors can consistently and concurrently update key-value pairs of the distributed state, which in turn enables lazy execution.

Overall, the Slash stateful executor tackles **C1** and enables highly efficient yet consistent processing model using RDMA-based building blocks. RDMA channels enable fast network transfer and address **C2**. Finally, Slash's distributed state backend solves **C3** via consistent state management, by a coherence protocol tailored to RDMA.

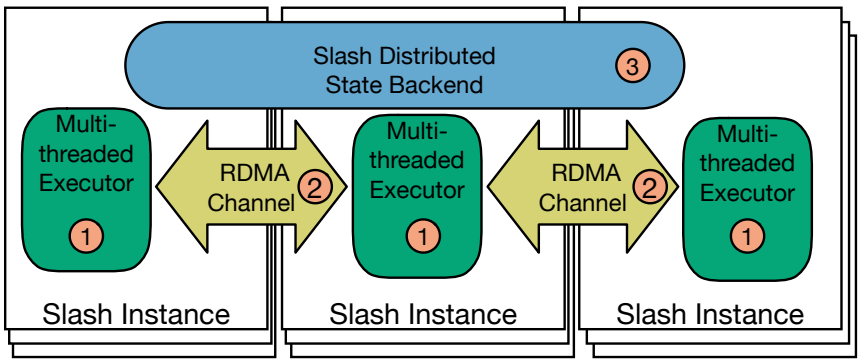


Figure 4.1: The Architecture of Slash.

4.4 Slash Stateful Executor

In this section, we present the Slash processing model (see Section 4.4.1) and discuss execution-related aspects: supported stream operators (see Section 4.4.2) and parallel execution (see Section 4.4.3).

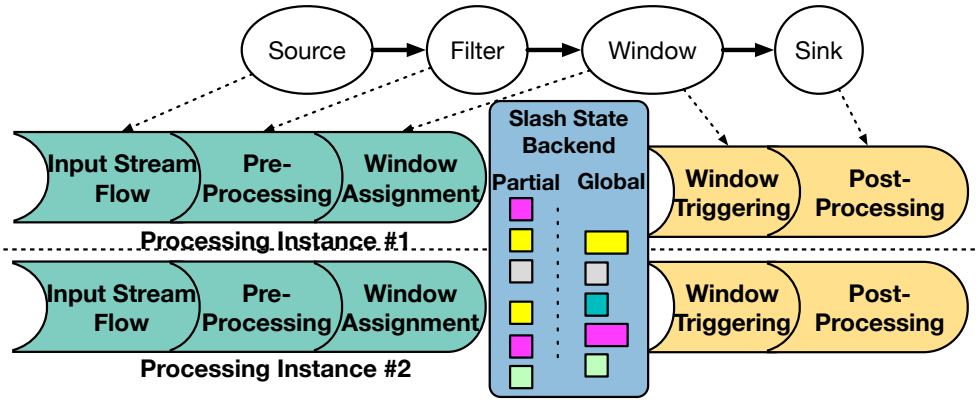


Figure 4.2: Slash translates a query comprising filter and a time-based windowing operators into pipelines (green and violet shapes). Slash executes each pipeline on its instances. Left pipelines update the distributed window state. Right pipelines trigger the window and read the distributed state.

4.4.1 Processing Model

Slash's stateful executor applies data-parallel transformations to physically partitioned data flows of a data stream. Thus, Slash runs in parallel multiple instances of the same operator across the nodes of a cluster. Slash does not assume that data flows are logically partitioned on the primary key, thus, a key may appear in multiple data flows.

Figure 4.2 depicts the stateful processing model of Slash. Slash supports stateless and stateful continuous operators as operator pipelines, in line with recent scale-up approaches to stream processing [47, 38, 5, 50]. Each pipeline terminates with a soft pipeline breaker, such as a window trigger operator [38, 5], as shown in Figure 4.2. However, Slash extends the late-merge scale-up approach to support scaling out. To this end, Slash shares operator state among a set of nodes (via RDMA) and omits data re-partitioning. Slash performs no data re-partitioning, e.g., no hash-based re-partitioning. Thus, operators immediately update the shared mutable state, which is kept consistent across Slash instances by the SSB. Furthermore, the degree of parallelism of a pipeline is bound by the number of its input data flows.

State and computation consistency involves two properties. We describe the two properties in the following and discuss below how we achieve them.

`Property 1 (P1).` Slash must not output any result at timestamp t that is computed using records bearing timestamps greater than t.

`Property 2 (P2).` A distributed computation over a data stream D in Slash must result after lazy merging in the same output that a sequential computation would produce processing D.

Progress Tracking. Scale-out SPEs rely on re-partitioning and in-band or out-of-band progress tracking to trigger event-time windows on a key basis [30, 21]. Slash omits data re-partitioning, which introduces a challenge in the progress tracking of the overall distributed computation. In fact, instances of a scale-out SPE that omits data re-partitioning must coordinate to detect window termination and merge partial windows. To satisfy *P1*, Slash relies on *vector*

clocks [135]. Every Slash executor e tracks the lowest watermark $l_{e,w}$ for each window w: the greatest event-time timestamp of the records that update the window. Upon lazy merging, Slash executors share among each other their low watermarks via RDMA to build a vector clock $V_w = \{l_{1,w}, \dots, l_{m,w}\}$, where m is the number of Slash executors. Through the vector clock, executors observe each other's progress and coordinate window triggering in event-time. Triggering occurs when a Slash executor determines a timestamp entry in the vector clock to be greater than the end timestamp of a pending window.

Consistency. Slash ensures computation consistency (*P2*) using an *epoch-based coherence protocol* [136, 137] and *conflict-free replicated data types* (CRDTs) [138]. While we discuss our coherence protocol in Section 4.6.2.2, we describe CRDT-related aspects in the following.

The state backend represents the partial state of a window as a CRDT. As a result, the window bucket (or the window slice) in Slash need to be represented as a CRDT. CRDTs enable merging partial state while guaranteeing consistent results as follows. A CRDT for a non-holistic window computation, such as an aggregation, relies on commutativity of the aggregation. A CRDT for a holistic window computation, such as a join, relies on join-semilattice and delta updates [139]. For instance, the CRDT for a sum-based window stores the partial sums of each parallel summation. Upon merging, the CRDT computes the final result as the sum of all partial values.

4.4.2 Stateful Operators

Slash provides two common stateful operators: hash-based aggregations and hash-based joins on event-time windows. Slash assumes windowing techniques that rely on window buckets [140] or general slicing [83], with the following modifications.

Windowing. Slash executes windowed operators as part of an operator pipeline that consists of a window bucket (or slice) assigner and a window trigger. The window assigner determines the bucket (or slice) to which a record belongs and updates it accordingly. In Slash, a window assigner does not assume pre-partitioned data, but offloads state consistency to the state backend. The window trigger outputs the window content based on event-time. In Slash, a window trigger requires a vector clock to evaluate the triggering condition and relies on the state backend to provide consistent state.

Windowed Aggregation. Slash provides hash-based aggregation that follows the *late merge* approach [47]. Each Slash executor thread eagerly computes its own local state, i.e., a partial hash-based aggregate for each in-flight window. In Slash, we scale-out *late merge* using RDMA acceleration and distributed CRDT-based aggregations.

Windowed Join. Slash offers a windowed streaming join based on a hash join. For every in-flight window, Slash eagerly builds a hash-table for the two streams based on the key and event-time of incoming records. When a window terminates, Slash probes the hash-tables to output per-key pairwise combinations of stored records. Slash ensures state consistency as the underlying distributed hash-table lazily concatenates all partial values with the same key.

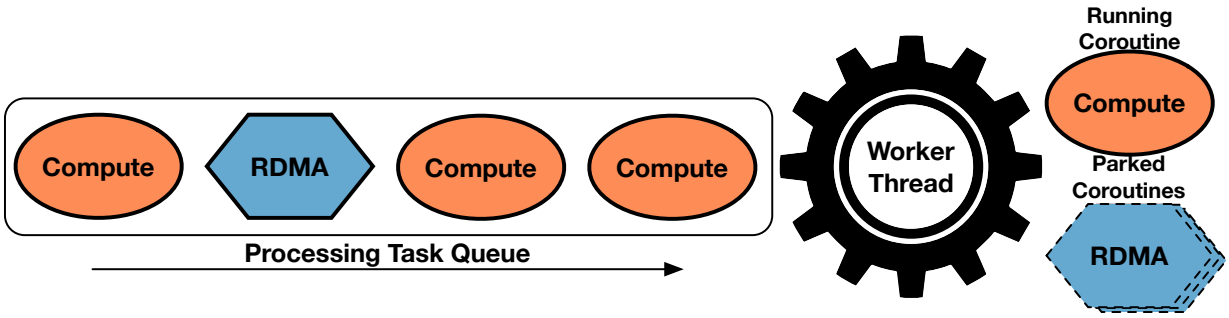


Figure 4.3: The coroutine-based event-driven scheduler of Slash.

4.4.3 Parallel Execution

The Slash executor parallelizes operators using worker threads across a number of nodes. For each physical operator, Slash assigns its RDMA channels to a worker thread. Each thread polls each channel for incoming data buffers to process. Slash uses an event-driven scheduler based on coroutines and a push-based processing model (see Figure 4.3). Coroutines are lightweight threads that enable cooperative multitasking [141]. The Slash scheduler interleaves compute coroutines with RDMA coroutines. RDMA coroutines execute RDMA-related tasks, such as polling buffers from an RDMA channel. Compute coroutines perform push-based processing on polled buffers. In the case of an empty RDMA channel, the scheduler parks the related RDMA coroutine and executes available ready compute tasks. Thus, empty RDMA channels do not stall the execution of pending compute coroutines.

We select coroutines, as they enable context switch with 10-20 ns of latency [129] and the interleaving of compute and I/O-tasks [142]. Current SPEs perform network-related operation on dedicated threads [30, 38, 5, 6]. However, performing RDMA operations on dedicated threads needs synchronization with processing threads, which wastes up to 400 cycles on common x86 CPUs [27]. The Slash scheduler hides network latency by executing compute tasks while RDMA packets are in-flight. This enables fine-grained control on RDMA and compute operations, which results in higher CPU efficiency.

Overall, Slash's processing model is inspired by LightSaber [5] and Grizzly [38], as it relies on task-based parallelism and late merging of partial state. However, Slash differs from them as follows. First, it extends the processing model of the above systems to target scale-out execution. To this end, it introduces eager, distributed computation of partial results and their lazy merge through RDMA. Second, it extends task-based parallelism using coroutines to interleave RDMA-related operations with processing tasks. Finally, it is agnostic to the execution strategy, as it supports compilation-based and interpretation-based strategies. Slash's worker threads have their own queues of coroutines to execute, whereas LightSaber and Grizzly share a single task-queue among their worker threads and focus on compilation-based execution.

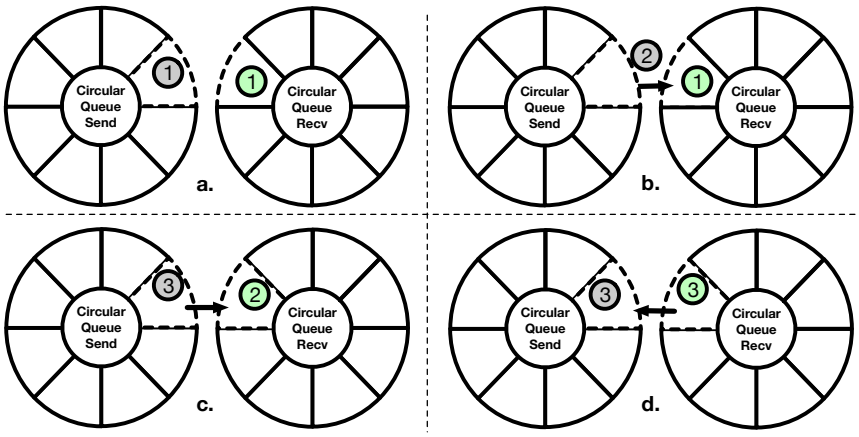


Figure 4.4: The protocol behind an RDMA channel: a.) the sender (SQ) acquires the dotted buffer, b.) begins the transfer, c.) waits for credit, and d.) acknowledge the completion on the receiver (RQ).

4.5 RDMA for Data Streaming

In this section, we present our RDMA-based transfer protocol that enables Slash to stream data with high throughput and low latency. We provide an overview of our protocol (see Section 4.5.1), describe its phases (see Section 4.5.2), and discuss details of our RDMA-channels (see Section 4.5.3).

4.5.1 RDMA Data Transfer Protocol

Our protocol determines the record exchange between a producer and a consumer via an RDMA channel. It defines the pipelined access to an RDMA-capable circular queue that guarantees FIFO delivery of records. Our protocol is consumer-driven: the consumer (based on its processing capabilities) requires the producer to adjust its sending rate to avoid back-pressure. Furthermore, it defines a coherence model to access the queue: a producer cannot overwrite unread buffers in the memory of the consumer. To this end, we schedule writes/reads to/from the queue via *credit-based flow control* (CFC) [143]. CFC is widely used in RDMA protocols to ensure coherence of RDMA-based data structures [129, 27]. In our solution, we use CFC to ensure FIFO delivery of records and avoid back-pressure.

4.5.2 Phases of the Protocol

Our protocol has two phases: a *setup* phase and a *transfer* phase. The setup phase defines the initialization of an RDMA channel, while the transfer phase defines the handling of data transfer at runtime.

Setup phase. This phase consists of 1) the initialization of a circular queue of RDMA-capable memory on the sender and receiver side and 2) setting up of a reliable RDMA connection

between the two parties. The circular queue has c slots, which is the initial number of credits. Each slot is an RDMA-capable, fixed-size buffer, which are allocated in this phase. The value of c is fixed throughout query execution, as its selection is hardware-sensitive and determines the level of pipelining, which depends on the NIC capabilities [52].

Transfer phase. Figure 4.4 shows the steps of the transfer phase. In this phase, a producer is permitted to: ① acquire the next buffer from the circular queue and write to it, ② post a write request for a buffer to an RDMA NIC, and ③ poll for credit from the consumer. A consumer is permitted to: ① poll for an incoming data buffer, ② mark the buffer for processing, and ③ send a credit to the producer.

Using pipelining, a producer that follows our protocol can send up to c buffers before it must wait for credit [27]. In particular, write requests do not overtake each other and result in a data buffer to be readable on the consumer upon their completion. To guarantee that a producer does not overwrite unread data, the consumer must notify the producer about writables buffer in its queue.

Properties. Based on the operations above, our protocol ensures three invariants. First, a producer decreases its number of credits by one after a write request. Second, a consumer transfers a credit to the producer after processing a buffer. This notifies the producer that the buffer is writable. Finally, a producer with no credit cannot pick buffers from the queue. As a result, it cannot push further write requests and has to wait for new credit from the receiver. Overall, producers and consumers that follow our protocol are guaranteed to consistently exchange records in FIFO order, at a self-adjusting data rate.

4.5.3 RDMA Channels

An RDMA channel consists of a QP, a circular queue, and a credit counter. RDMA channels enable zero-copy transmission and reception of buffers using RDMA. Fundamental choices behind this component are a flat memory layout of the circular queue and a push-based transfer model via RDMA WRITEs. The choices influence design regarding data structure, RDMA verbs, and message layout.

Data structure. The circular queue consists of an RDMA-capable memory area of $c \times m$ bytes, with c as number of credits and m as the size of a single buffer (see Figure 4.5). As a result, buffers are contiguously stored, which induces a flat memory layout. Each buffer comprises of contiguous payload and metadata, such as a flag for polling. A flat layout is beneficial for three reasons. First, it avoids expensive pointer chasing operations [47]. Second, contiguously-stored payload and metadata enable data transfer via a single RDMA request, whereas decoupled data region and metadata would require two RDMA requests. Finally, it allows for cacheline alignment and huge-pages allocation, which reduce CPU cache misses and NIC TLB misses [52].

RDMA verbs. We select a push-based transfer approach using RDMA WRITEs instead of RDMA READs for the following reasons. First, an RDMA READ involves a round-trip per message, which leads to higher latency and CPU utilization [129]. In contrast, an RDMA WRITE needs a single trip per message. Second, RDMA WRITEs enable push-based transfer:

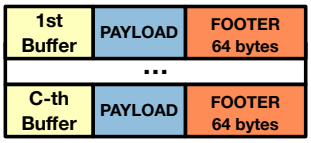


Figure 4.5: The layout of a circular queue with c buffer entries.

the producer writes into the memory of the consumer, which polls its local memory. In contrast, RDMA READs allow for pull-based transfers: the consumer continuously reads the producer's remote memory until the requested data is available. Thus, an RDMA pull-based model induces extra network traffic, as polling occurs over RDMA. Overall, our push-based approach requires only one network access per message and efficiently polls local memory.

Message layout. Slash transfers buffers as messages via RDMA WRITEs. This needs a detection mechanism of inbound messages at the receiver. To this end, we divide the buffer into a data region for the payload and a footer for metadata. We use the final byte of the footer for polling, which has two benefits compared to polling on the header. Polling on the footer guarantees full data transfer, as RDMA WRITE transfers buffers from lower to higher memory addresses. Polling on the header does not ensure full reception of a buffer, as the transfer might still be in progress. The consumer can safely process the data region when it detects the change on the last byte.

4.6 Slash State Backend

The Slash State Backend (SSB) is a concurrent key-value store for in-memory operator state. It provides state management techniques to build global operator state shared across multiple nodes using RDMA. In this section, we describe our approach to RDMA-accelerated state management (see Section 4.6.1) and the components of our SSB (see Section 4.6.2).

4.6.1 RDMA-accelerated State Management

In this section, we describe our approach to accelerate state management of a scale-out SPE using RDMA. Our state management leverages RDMA to enable the nodes of an SPE to consistently read and write each other's state with high bandwidth and low latency. To this end, we first present requirements for a state management component (see Section 4.6.1.1) and then discuss the design of the SSB (see Section 4.6.1.2).

4.6.1.1 Requirements

State management defines how operators access and modify state. To speed-up state access via RDMA and enable running operators to concurrently modify shared state, we analyze requirements on workloads and RDMA semantics.

Workload. A state backend for SPEs has three strict design requirements [134]. First, a state backend must support update-intensive workloads as stateful operators concurrently perform

point updates of the state on a record basis. Point updates consists of *read-modify-write* (RMW) operations that change a key-value pair based on the previous value and the record content. Second, a state backend must enable efficient scans of its content, for example, to timely trigger and post-process a window. Finally, it must allow for arbitrary state sizes, which may exceed single-node memory boundaries.

RDMA semantics. A state backend must efficiently handle concurrent updates among nodes. Nodes may concurrently update the same key-value pair and thus the state backend must ensure consistent update semantics. This is a two-fold challenge: 1) needs a coherence protocol among nodes to achieve consistency and 2) involves careful design of memory access patterns from the local CPU as well as remote RDMA NICs, as they are not coherent.

4.6.1.2 Design of the Slash State Backend

Based on the above requirements, we consider the following design choices to achieve RDMA acceleration for consistent state management. As discussed in Section 4.4, Slash does not perform re-partitioning but uses shared mutable state for stateful operators. Shared mutable state enables concurrent reads and writes on the same key-value pair. However, this requires expensive coordination among readers and writers, which we avoid with our SSB as we show in the following.

Partial State. SSB maintains on every executor a partial state for each locally-running operator (see Figure 4.6). Operators eagerly update partial state locally, which is in line with common scale-up principles [47, 38, 5]. With our approach, the common operation is the per-record update of partial state, which neither induces queueing among operators nor suffers from skew-sensitive hash partitioning [47]. In contrast, the common operation for traditional SPEs is the per-record partitioning and the update of co-partitioned state (see Figure 4.7).

State Maintenance. The SSB divides the key-value space into disjoint partitions and assigns each partition to an executor. Each executor is the *leader* for only one partition, which we call *primary* partition. Slash does not perform re-partitioning thus each executor potentially maintains state that belong to the partition of another leader. Thus, an executor stores a *fragment* of each remote primary partition and becomes *helper* of their respective leader executors. For each partition, its leader and helpers synchronize their content based on the following coherence protocol. This leads to a space amplification for key-value pairs proportional to the number of nodes. However, the value size depends on the semantics of the window computation. Thus, non-holistic window results in an aggregate per node, whereas holistic windows results in disjoint sets of values per node.

Coherence Protocol. The SSB lazily synchronizes partitions between leader and helper executors using an epoch-based coherence protocol. An epoch is a time span between two synchronization points. At the end of an epoch, helper nodes of a partition send its content to its leader node, which merges key-value pairs. Furthermore, this enables arbitrary sizes of state as it is scattered across aggregated memory of the cluster and is materialized only at the end of an epoch.

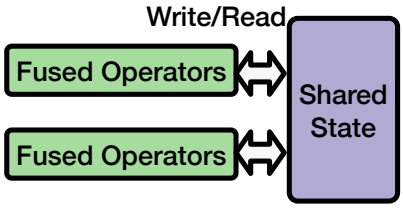


Figure 4.6: Shared State.

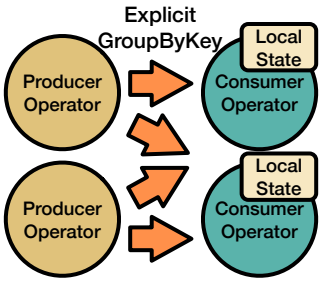


Figure 4.7: Data Repartitioning.

Update conflicts. The SSB needs to support concurrent updates of the same key-value pair from multiple executor. To this end, Slash relies on CRDT to merge conflicting key-value pairs (see Section 4.4.2). Our SSB enables a processing model that omits data re-partitioning and makes common case operation fast. The common case operation of Slash is the eager computation of partial state, while current SPEs partition records prior to update state.

4.6.2 Components of Slash State Backend

The SSB is our state storage layer, which provides distributed hash tables to consistently manage operator state. In the following, we describe the techniques behind our SSB: our distributed hash table (see Section 4.6.2.1) and our epoch-based coherence protocol (see Section 4.6.2.2).

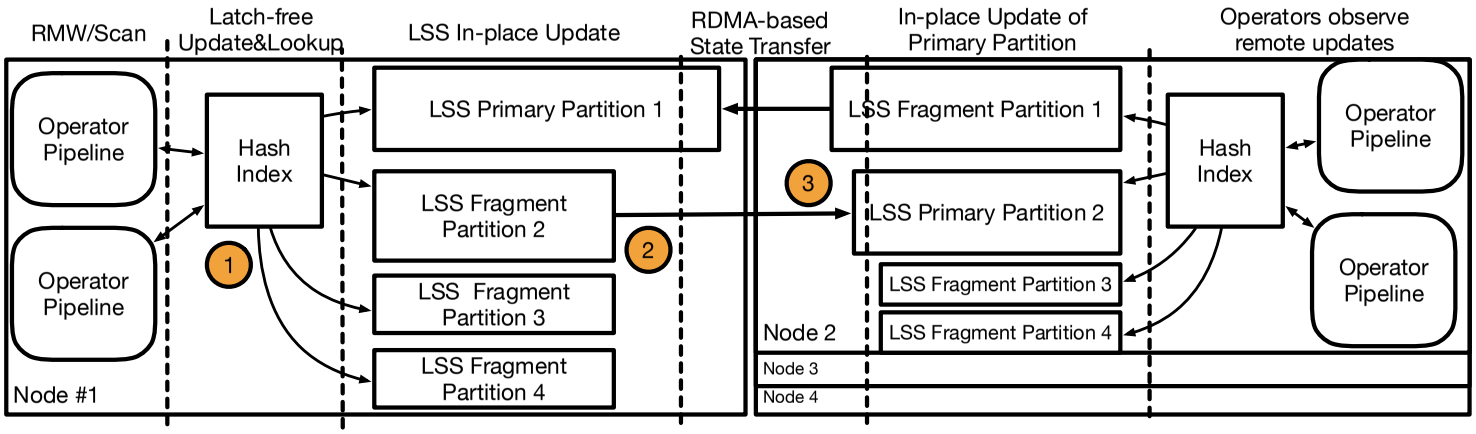
4.6.2.1 Distributed Hash Table

The SSB uses a distributed hash table based on separate chaining and log-structuring. The hash table consists of a *hash index* and a *log-structured storage* (LSS) [144, 145, 146] of key-value pairs for each partition. We show the architecture of our distributed hash table in Figure 4.8a. In the following, we provide the rationale behind our design and describe the LSS.

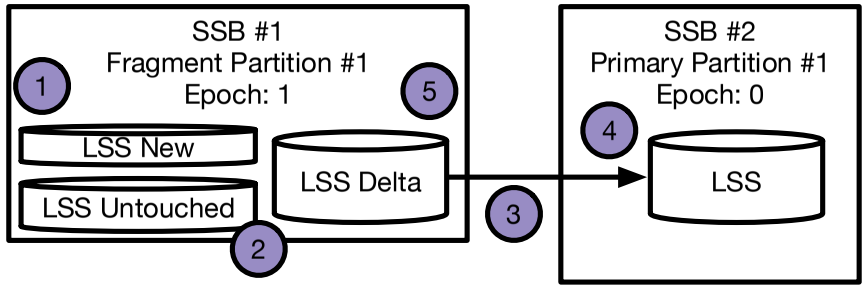
Rationale. An update of a key-value pair requires a lookup in the hash index to find the position of the pair in one of the LSSs. Decoupling indexing from storage has two advantages over techniques such as open addressing [147]. First, it enables one index for each partition that points to multiple LSSs (1). Second, log-structuring induces temporal locality for the updates, i.e., frequently accessed key-value pairs are in the same portion of the log. This enables quick detection of changes in the LSS so that a helper can send them to a leader using RDMA without involving pointer chasing. As a result, a helper moves to a leader only the last modified pairs to avoid redundant network transfer. In contrast, open addressing induces a scattered memory layout that requires a full scan to detect update. Finally, we do not assume a particular design for the hash index. Instead, we use the hash index of FASTER [134] in the remainder of this Chapter.

Log-structured storage. Our LSS is an RDMA-capable circular buffer that stores dense key-value pairs. We partially follow the design of FASTER [134] and consider its in-memory capabilities. However, we extend its design to enable RDMA acceleration in a distributed setting

but skip disk spilling, as it is out of our scope. The LSS acts as a hybrid log that enables concurrent append and in-place update operations on key-value pairs (2). We extend FASTER's design as follows. We rethink its design for distributed execution and introduce leader and helper nodes. Helper nodes transfer delta changes to leader executors in chunks using dedicated RDMA channels. Slash interleaves reception and merging of delta changes with query processing. Furthermore, we enable the circular buffer to adaptively resize as partitions vary in size over time due to frequency shifts in key distributions. Consequently, our state backend adapts to shifts in workloads size. Overall, our state backend enables incremental state synchronization via our epoch-based coherence protocol (3).



(a) Internals of our Distributed Hash Table: operator pipelines transparently modify state while the SSB ensures its consistency.



(b) The Epoch Protocol: SSB#1 sends the delta of the fragment partition #1 at epoch 1 to SSB#2 over RDMA.

Figure 4.8: The Slash State Backend: an overview of its architecture and a detail of the epoch protocol for state consistency.

4.6.2.2 Epoch-based coherence protocol

Slash slices infinite streams into finite chunks of records based on epochs. Epoch-based concurrent systems safely execute global operations at epoch boundaries. Many systems rely on epoch-based synchronization for diverse goals, such as checkpointing [30, 134]. In Slash, we extend the concept of epochs to merge distributed shared partitions lazily (stored on helper nodes) into their respective primary partition. The SSB follows an epoch-based coherence protocol that enables nodes to synchronize state and ensures consistency. In the following, we present the setup and synchronization phases as well as the properties of our protocol.

Setup Phase. Consider a Slash deployment of n Slash Executors and n primary partitions, where n is the number of nodes. Each partition has an epoch counter to version its content. In the setup phase, each leader executor connects to all possible executors. Overall, Slash creates n^2 RDMA channels for state synchronization during this process. Note that our RDMA channels for state transfer use the LSS memory instead of dedicated circular queues to avoid data copies. Slash assumes epoch duration to be agnostic to window size. However, a Slash instance signals the ahead-of-time termination of an epoch upon window triggering.

Synchronization Phase. We assume that each stateful operator receives records as well as tokens that notify system-wide events, such as punctuations. This is a common technique used in several SPEs to make operators perform operations, e.g., trigger windows or take a state snapshot [30]. In Slash, the arrival of a synchronization token to an operator makes helpers perform the following steps, which we show in Figure 4.8b.

1. Increment the epoch counter for each shared partition.
2. Identify the portion of the circular buffer that contains the latest changes in the LSS of each modified partition. Prior to the transfer, mark the changes as read-only to prevent inconsistency between DMA reads and CPU writes.
3. Transfer the changes in the circular buffer via RDMA channels.
4. Incrementally merge the transferred content in the local LSS.
5. After the transfer, invalidate the content of the transferred portion of the storage so that it can serve further RMW operations.

Properties. In response to the above steps, leader executors lazily receive updates for the state they manage. We piggyback vector clock updates with state updates so that a leader executor can observe the progress of helpers. A leader node can trigger a per-key window at timestamp t only if the vector clock guarantees the occurrence of no record nor state update that bears an event-time timestamp smaller than t. Note that a local epoch counter induces an order on the arrived updates such that state updates cannot skip each other. Furthermore, discarding transferred content is safe, as RMW operations restart from a zero value.

Distributed instances of the SSB that follow this protocol are guaranteed to converge to a consistent state at the end of each epoch. Window operators benefit from this approach as the triggering of a window occurs at the end of an epoch. As a result, the window state becomes consistent upon triggering, which ensures correct results.

4.7 Evaluation

In this section, we experimentally validate the system design of Slash through a set of end-to-end experiments and micro-benchmarks. First, we describe the setup of our evaluation in Section 4.7.1. Second, we compare Slash against RDMA UpPar, LightSaber, and Apache Flink on end-to-end queries (see Section 4.7.2). Third, we perform a drill-down analysis on Slash and RDMA UpPar to understand the implications behind our design choices (see Section 4.7.2). Finally, we sum up our key findings of our evaluation in Section 4.7.4.

4.7.1 Experimental setup

In the following, we introduce our hardware and software configurations (see Section 4.7.1.1) as well as the selected workloads (see Section 4.7.1.2).

4.7.1.1 Hardware and Software

In our experiments, we use the following hardware and software configurations.

Hardware Configuration. We run the experiments on an in-house, 16-node cluster. Each node is equipped with a 10-core, 2.4 Ghz Intel Xeon Gold 5115 CPU, 96 GB of main-memory, and a single-port Mellanox Connect-X4 EDR 100Gb/s NIC. Each NIC is connected to a 100 Gbits InfiniBand EDR switch by Mellanox. Every node runs Ubuntu Server 16.04. We disable hyper-threading and pin each thread to a dedicated core. Unless stated otherwise, every hardware component is configured with factory settings.

Software Configuration. In our evaluation, we use Slash, RDMA UpPar, LightSaber [5], and Apache Flink 1.9 [35] as Systems under Test (SUTs). We select Apache Flink as a representative of production-ready, scale-out SPEs based on managed runtimes, whereas we choose LightSaber as representative of scale-up SPEs. Flink provides queue-based partitioning to scale-out query processing. To configure Flink, we follow its configuration guidelines [148]. On each node, we allocate half of the cores for processing and the other half for network I/O. We reserve 50% of the OS memory to Flink and allocate the remaining memory to store the input dataset that we stream via main memory. We configure Flink to use IPoIB on our RDMA cluster.

We build Slash with `O3` compiler optimization and native CPU support using `gcc 9.3`. We configure Slash to use all physical cores and 48 GB of memory (using 2MB hugepages) for RDMA-related operations. Unless stated otherwise, we run Slash with the best configuration parameters that we present in Section 4.7.3 and configure the epoch of SSB to end every 64 MB of data. We follow similar configuration steps for LightSaber and RDMA UpPar. Note that we use Slash's RDMA channel to implement RDMA UpPar.

4.7.1.2 Workloads

To experimentally validate our system design, we select the *Yahoo! Streaming Benchmark* (YSB) [47], the *NEXMark benchmark suite* (NB) [149], and the Cluster Monitoring benchmark

(CM) [150]. We choose YSB and NB, as they are commonly-used benchmarks that represent real-world scenarios [47, 151]. We select CM, as it is based on a publicly-available, real-world dataset provided by Google. Furthermore, we introduce a self-developed *Read-Only* (RO) benchmark for our drill-down analysis.

YSB. The YSB assesses the performance of windowed aggregation operators. A record is 78-bytes large and stores an 8-bytes primary key and an 8-bytes creation timestamp. YSB consists of a filter, projection, and a time-based, per-key window. Following YSB specifications, we use a 10m event-time, tumbling count window.

NB. The NB simulates a real-time auction platform with three logical streams: an auction stream, a bid stream, and a seller event stream. Records are 206 (seller), 269 (auction), and 32 (bid) bytes large. Each record stores an 8-bytes primary key and an 8-bytes creation timestamp. The NB contains queries with stateless and stateful operators.

We use queries 7 (NB7), 8 (NB8), and 11 (NB11) to cover a wide range of scenarios. Based on these queries, we define three workloads to assess our SUTs. NB7 contains a window aggregation with a window of 60s on the bid stream. We select NB7, as it features small state sizes and an RMW state update pattern. NB8 consists of a 12h tumbling window join in event time over the auction and seller streams. We choose NB8, as it reaches large state sizes due to its append pattern for state update and large tuple sizes. NB11 consists of a session window join in event time over the bid and seller streams. We choose NB11 to assess the effect of small tuple size on the join implementation. We omit the other queries in the suite, as they are either stateless (NB1-2) or evaluate aggregations and joins (NBQ3-14), which we cover already with the selected queries.

CM. The CM benchmark executes a stateful aggregation over a stream of timestamped records containing the traces from a 12.5K-nodes cluster at Google. Each record is 64 bytes large and stores an 8-bytes primary key and an 8-bytes timestamp. The stateful aggregation is a 2s tumbling window that computes the mean CPU utilization of each executed job.

RO. The RO benchmark is a stateful query that counts the number of occurrences of items in a stream. We implement RO to investigate I/O bottlenecks, as data flows throughout the system without any costly computation. Each record stores an 8-bytes primary key and an 8-bytes creation timestamp. A stateful operator maintains the count of occurrences of each key. Keys are drawn following uniform distribution from a 100M-wide range.

Experiment Overview. We structure our evaluation as follows. First, we execute end-to-end queries to compare Slash, RDMA UpPar, LightSaber, and Flink (see Section 4.7.2). To this end, we scale the input data size up to the number of nodes to perform weak scaling experiments [152]. Second, we run a series of micro-experiments to reason in detail about the performance behavior of Slash components (see Section 4.7.3). Specifically, we breakdown the execution time of Slash and RDMA UpPar to perform a micro-architecture analysis [153] to reveal how well Slash uses hardware resources. Finally, we summarize the key findings of our evaluation (see Section 4.7.4).

4.7.2 End-to-end Queries

In this section, we focus on the execution of end-to-end queries by the SUTs. We present our evaluation methodology in Section 4.7.2.1. We evaluate queries with windowed aggregations (YSB and NB7) in Section 4.7.2.2 and queries with windowed join (NB8 and NB11) in Section 4.7.2.3. Finally, we conduct in Section 4.7.2.4 a COST (Configuration that Outperforms a Single Thread) analysis [154] and compare Slash against LightSaber [5], which is an SPE optimized for single-node execution.

4.7.2.1 Methodology

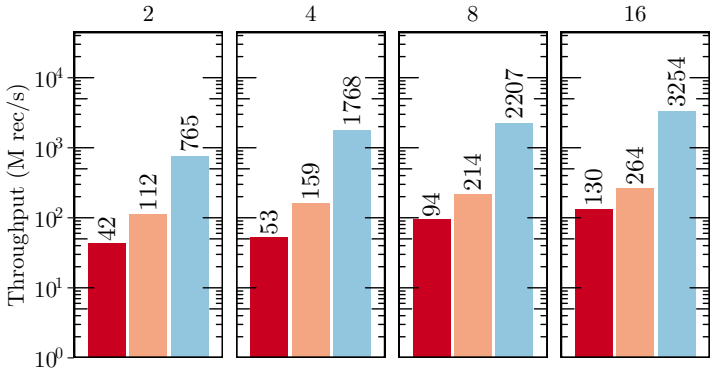
In our end-to-end evaluation, we follow the benchmark methodology proposed in earlier research [47]. We pre-generate the dataset to stream data from main memory, to omit record creation and ingestion overhead. Thus, the upper bound for the input rate is the main-memory bandwidth. In every run, source operators consume data in real-time, which SUTs process. During execution, we measure query processing throughput, which we define as the number of records the SUT can process in one second. We repeat each experiment multiple times and compute average measurements. Through our experiments, we evaluate the efficiency of the SUTs while they execute a query.

4.7.2.2 Queries with Windowed Aggregations

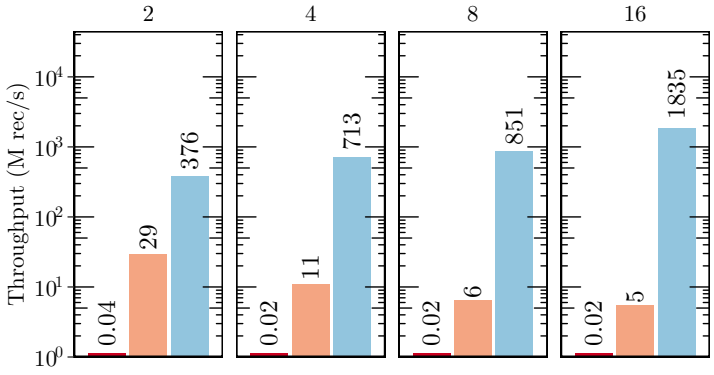
In this section, we focus on the performance comparison between the SUTs while they perform windowed aggregations using YSB, CM, and NB7.

Workload. In YSB and NB7, each executor thread processes a partition of 1 GB of input data. In CM, each executor thread processes a partition of the provided input dataset. Each YSB partition contains records with a primary key drawn uniformly from a 10M-wide range. Each NB7 partition contains bid records with a primary key generated following a Pareto distribution that induces a long-tail due to heavy-hitters. Partitions of YSB, NB7, and CM are non-disjoint: the same key can occur multiple times in multiple partitions. In addition, we scale the number of executor threads and nodes. We configure each SUT to use 10 threads per node and up to 16 nodes. However, RDMA UpPar and Flink need to partition the input stream before the window operator. As a result, they use half the threads to execute the filter and projection and the second half for the window operator. Instead, Slash runs filter, projection, and windowing on all threads.

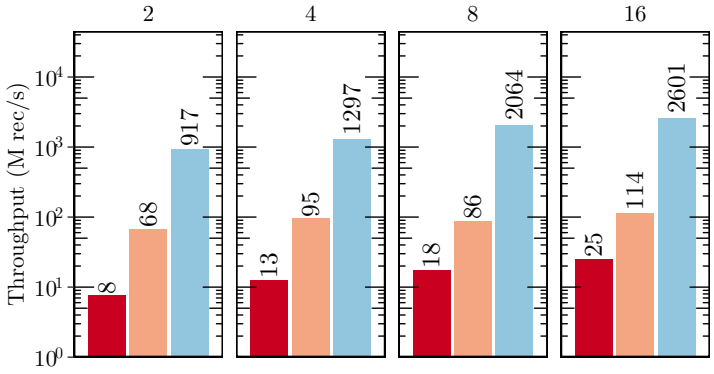
Result. In all YSB, CM, and NQB7 experiments, Slash outperforms all SUTs (see Figure 4.9a, 4.9b, and 4.9c). It achieves up to 12x and 25x higher throughput on the YSB compared to RDMA UpPar and Flink, respectively. Slash attains up to 22x and 104x higher throughput on the NQB7 compared to RDMA UpPar and Flink, respectively. Similarly, Slash achieves up to two order of magnitude higher throughput than RDMA UpPar and Flink, while executing CM. Overall, Slash is the only SUT to achieve throughput of up to 2 billion records/second and almost linear weak scaling in YSB , CM, and NB7.



(a) YSB.



(b) CM.



(c) NB7.

Figure 4.9: Throughput for queries including a windowed aggregation: YSB, CM, and NB (in records/s) executed on Flink (●), RDMA UpPar (●), and Slash (●) on 2, 4, 8, and 16 nodes.

Discussion. In this experiment, Slash achieves almost linear weak scaling, whereas other SUTs result in sub-optimal performance. The reasons for this superior performance for windowed aggregations are two-fold. First, as shown by previous research [47], queue-based partitioning

of input records introduce a significant bottleneck in single-node setups. This also applies in the distributed case, as network transfer is mediated by software queues. As a result, SUTs that use queue-based partitioning to scale-out incur an inherent bottleneck regardless of the network hardware. Second, Slash efficiently computes local partial states and consistently merges them using point-to-point RDMA transfers among nodes. In contrast, Slash is not affected by a performance regression, when processing workloads that have a skewed distribution of partitioning keys. Overall, Slash attains higher throughput due to better utilization of underlying hardware resources, as we further explain in Section 4.7.3.

4.7.2.3 Queries with Windowed Joins

In this section, we focus on the performance of our SUTs as they run windowed joins of NB8-11.

Workload. We follow the same setup of NB7 except for the following aspects. Each executor thread processes a partition of 1 GB of input stream, in which the ratios between auction and seller and between bid and seller are 4 to 1, according to the benchmark. Note that every bid has always a valid seller.

Result. In all NQB8 and NB11 runs, Slash achieves higher throughput compared to the other SUTs (see Figure 4.10a and 4.10b). In NQB8, it reaches up to 8x and 128x higher throughput than RDMA UpPar and Flink, respectively. In NQB11, it shows up to 1.7x and 40x higher throughput than RDMA UpPar and Flink, respectively.

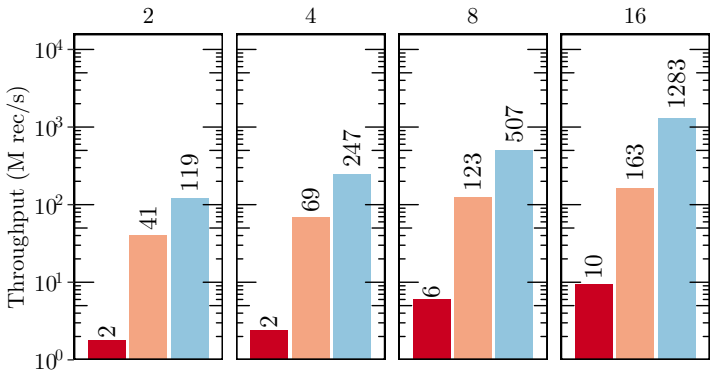
We observe that Slash achieves almost linear weak scaling on queries with join operators, similarly to windowed aggregation. In contrast, Apache Flink and RDMA UpPar exhibit a severe loss in throughput. In sum, Slash does not achieve the same performance gain of aggregations, although it outperforms the other SUTs.

Discussion. The main performance differences for windowed joins among the SUTs result from the following characteristics. First, a windowed join operator is more memory-intensive than a windowed aggregation. A hash-based streaming join operator appends every record to intermediate state including the records that have no matching join partner yet. As a result, append operations in Slash do not benefit from CPU cache temporal locality. In contrast, RMW-based aggregations benefit from CPU caches, as the RMW operations induce temporal relations to cache accesses. Second, partitioning in RDMA UpPar and Flink induces performance regression as in the case of windowed aggregation. This becomes more severe while increasing the number of nodes. In contrast, Slash does not show performance regression when scaling out.

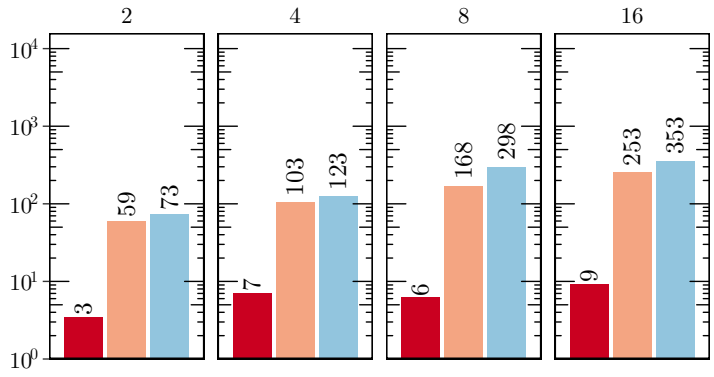
In sum, hash-based streaming joins do not show the same performance gain as windowed aggregations, as join are limited by compute resources. We plan to conduct further studies on the RDMA-based acceleration of streaming joins as future work.

4.7.2.4 COST Analysis

In this experiment, we compare the processing performance of Slash against a scale-up SPE, following the COST metric proposed by McSherry et al. [154]. We select LightSaber as the



(a) NB8.



(b) NB11.

Figure 4.10: Throughput for queries including a windowed join: NB8 and NB11 (in records/s) executed on Flink (●), RDMA UpPar (●), and Slash (●) on 2, 4, 8, and 16 nodes.

latest proposed scale-up SPE, which does not run on a managed runtime (as BriskStream [6]). We choose CM, NB7, and YSB as workloads supported by both SUTs, as LightSaber does not support joins. In Figure 4.11, we show the throughput of LightSaber (L) and of Slash (on 2, 4, 8, and 16 nodes) on the selected workloads.

We observe that Slash outperforms LightSaber in each run as it improves its performance when doubling the number of nodes. Furthermore, Slash achieves almost linear speedup on YSB and CM (up to 11.6x throughput increment using 16 nodes) and sub-linear scaling on NB7 compared to LightSaber (up to 4.4x throughput increment using 16 nodes), respectively. The improvement of Slash over LightSaber is smaller compared to the gain over RDMA UpPar, as LightSaber's execution is agnostic to data re-partitioning. Overall, the key insight of this experiment is that LightSaber offers a valid alternative to SPEs that rely on data re-partitioning as long as the workload 1) is sustainable on a single node, 2) does not need RDMA ingestion, and 3) does not involve join operators. For highly-demanding workloads, Slash offers robust scale-out performance, as our evaluation shows.

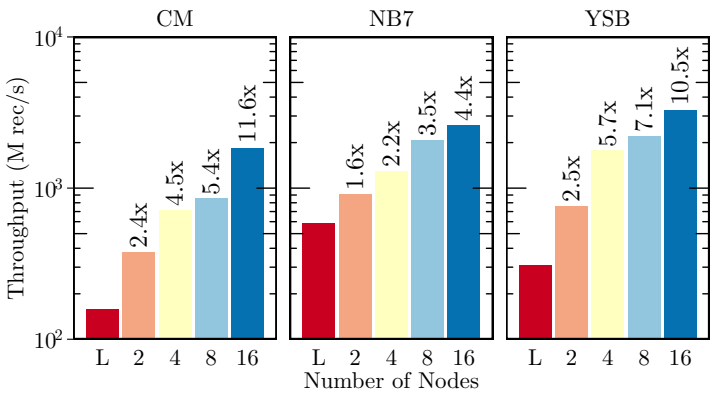


Figure 4.11: COST comparison against LightSaber (L).

4.7.3 Performance Drill-down

In the previous section, we have analyzed the performance result of Slash and Flink on end-to-end queries. In this section, we reveal the reasons behind the improvement in performance of Slash over RDMA UpPar. We omit Flink in this evaluation, as its partitioning approach suffers from runtime and IPoIB overhead [47]. In the following, we first describe the methodology for our drill-down evaluation in Section 4.7.2.1. After that, we assess in Section 4.7.3.2 the maximum achievable throughput of RDMA UpPar and Slash in our RDMA evaluation setup. In this setup, we consider application-related aspects, such as parallelism and data skewness. In Section 4.7.3.3, we break down the execution time of both SUTs to analyze the impact of each CPU component. Finally, in Section 4.7.3.4, we analyze the resource utilization of each SUT on a stateful workload using hardware performance counters.

4.7.3.1 Methodology

In our performance drill-down, we analyze workload-related and hardware-related aspects of Slash and RDMA UpPar. Workload-related aspects provide a high-level identification of bottlenecks and include data characteristics and application settings. Hardware-related aspects consist of hardware performance counters that we use to conduct micro-architecture analysis. These metrics allow us to derive the CPU components that stall the execution and the saturation point of the RDMA links. In the following, we consider the RO and YSB benchmarks and provide a brief description of the sampled metrics for each class of experiments.

4.7.3.2 Analysis of workload-related aspects

In this section, we compare the performance of RDMA UpPar and Slash with a focus on RDMA-based data transfer. We consider the RO query, which is primarily I/O bound, to evaluate the impact on performance of data re-partitioning. We analyze the effect on throughput and latency during query processing of application-related knobs, such as parallelism and buffer size, as well as data characteristics

Workload. We setup two Slash instances on two servers connected by a single RDMA NIC to measure the impact of buffer size on throughput and latency. The producer instance streams the input data to the consumer instance via our RDMA channels. The consumer instance polls the RDMA channels and applies stateful operator logic based on the benchmark. Each instance uses up to 10 threads for the executor. Every producer thread on the first node sends buffers of records via RDMA to one consumer thread on the other node in Slash. In RDMA UpPar, every producer thread sends buffers of records to any consumer via hash-partitioning. To measure the impact of parallelism on throughput, we use up to 8 nodes. Note that we configure our RDMA channels to use $c = 8$ (credits). Other configurations, such as $c = 8$ and $c = 16$ decrease throughput by up to 3%, whereas $c = 64$ leads to a performance regression by up to 10%.

Results. In this experiment, we assess the impact of application-related aspects on the performance of both SUTs. First, we show the impact of buffer size on throughput for both SUTs (see Figure 4.12a) and latency (see Figure 4.12b). Second, we measure the effect of parallelism (see Figure 4.12c) by scaling the number of threads and nodes. In Figure 4.12a and 4.12c, we mark in red the maximum achievable network bandwidth (11.8 GB/s), which we measure using the *ib_write_bw* tool [155]. Finally, we analyze the impact on throughout of a skewed distribution of the partitioning key (see Figure 4.12d). To this end, we generate partitioning keys following a Zipfian distribution using $z = 0.2 \dots 2.0$.

`Throughput.` We observe that Slash outperforms RDMA UpPar in all configurations as it utilizes up to 95% of the available network bandwidth (11.2 GB/s out of 11.8 GB/s) using two threads. Slash almost saturates the theoretical bandwidth limit of one RDMA NIC using two threads and 32 KB buffer size. In contrast, RDMA UpPar utilizes up to 50% of the available network bandwidth, i.e., 5.9 GB/s.

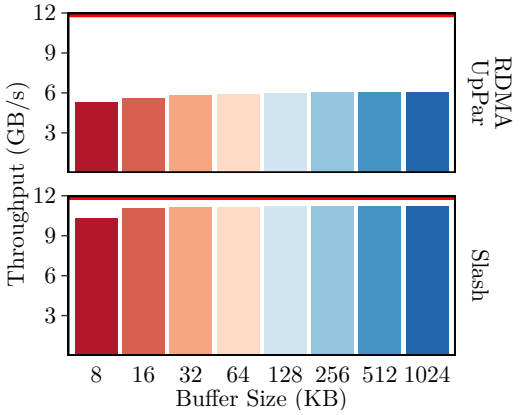
`Latency.` Slash achieves latencies below 100 μs for buffers sizes below 128 KB, while it achieve up to 1 ms of latency with 1 MB buffer size and above. In contrast, latencies of RDMA UpPar for each buffer size are about 10% higher than Slash.

`Parallelism.` Slash achieves the highest aggregated throughput for query processing (see Figure 4.12c). Slash achieves 11.2 GB/s on the RO benchmark using two threads. In contrast, RDMA UpPar requires 10 thread to saturate up to 91% of the available network throughput.

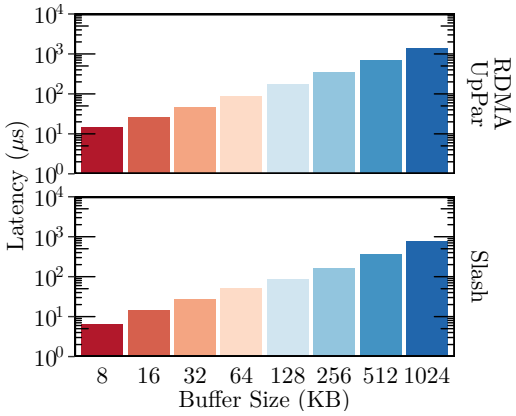
`Data Skewness.` Slash shows robust performance in the presence of a skewed distribution of the partitioning keys. Interestingly, we observe that the throughput of Slash increases, when the partitioning keys in the data stream are highly skewed. In contrast, the throughput of RDMA UpPar decreases by up to 68% (RO) and 110% (YSB), while the skewness in the distribution increases.

Discussion. Overall, our experiments show three interesting aspects. First, Slash becomes network bound with a lower number of threads compared to RDMA UpPar (i.e., 2 vs. 10). This induces an important benefit: increasing the number of threads and RDMA NICs per node results in higher processing throughput. We cannot derive the same conclusion for RDMA UpPar, as it requires an higher number of threads to achieve almost full line rate. As a result, Slash exhibits a higher per-thread efficiency compared to RDMA UpPar. Second, buffer size plays an

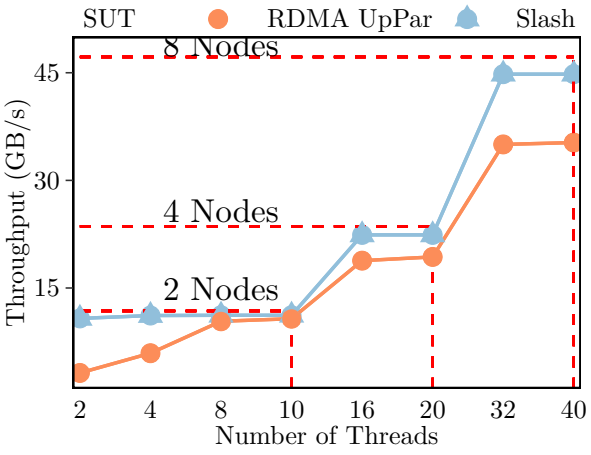
interesting role also for data stream processing on RDMA hardware. It enables the highest (or lowest) throughput (or latency) based on workloads and service constraints. In particular, both SUTs achieve micro-second latency, which is one order of magnitude lower than the latencies measured on Flink (not shown in the figures). Finally, Slash offers more robust performance than RDMA UpPar, in the presence of skewed data. RDMA UpPar suffers a performance regression, as hash-partitioning causes load imbalance due to the data-dependent selection of the consumer induced by data skewness. In contrast, Slash achieves constant throughput on RO, regardless of the skewness, as the transfer performance of RDMA channels is not data-dependent. When executing a stateful query, such as YSB, skewness results in higher throughput for Slash, as it reduces the number of key-value pairs of the state to be merged by the SSB.



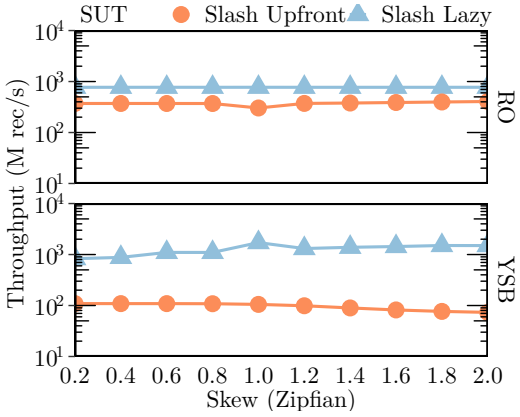
(a) Effect of buffer size on throughput.



(b) Effect of buffer size on latency.



(c) Effect of parallelism on throughput.



(d) Effect of skew on throughput.

Figure 4.12: Drill-down analysis of Slash and RDMA UpPar. We consider the impact on throughput and latency of buffer size (a, b), parallelism (c), and skewness in the distribution of the partitioning key (d).

4.7.3.3 Execution Breakdown

In the previous section, we have analyzed the effect of application-related knobs on throughput and latency of query processing. In particular, we have shown that RDMA UpPar provides lower efficiency compared to Slash. In this section, we perform an execution breakdown to reveal the reasons behind the sub-optimal performance of RDMA UpPar in comparison to Slash. To this end, we carry out a micro-architecture analysis of Slash and RDMA UpPar on the RO benchmark.

Metrics. Before delving into our analysis, we provide a brief description of the considered metrics. On a high level, recent x86 CPUs consist of two pipelined components: a *front-end* and a *back-end*. The front-end decodes instructions into μ-*ops* and delivers up to four μ-ops per cycle to the back-end. The back-end processes μ-ops out-of-order by allocating execution units and loading data from memory. Completed μ-ops are defined as *retired* (R) and constitute the useful work performed by a CPU. Front-end stalls, back-end stalls, and branch mis-prediction are sources of inefficiency for a CPU. Upon a front-end stall, the back-end has no μ-ops to process, thus, the application is front-end bound (FeB). Upon a back-end stall, a μ-ops needs to wait for data from the memory subsystem or for an execution unit. In the former case, the execution is *Memory-bound* (MemB), whereas in the latter case, it is *Core-bound* (CoreB). Finally, *bad speculation* (BadS) due to branch mis-prediction results in the cancellation of μ-ops prior to their retirement.

Workload. We execute the RO benchmark with best configurations of both SUTs that we derive in the previous experiments and refer to the nodes as sender and receiver. Specifically, we choose 64 KB as buffer size and repeat the measurements using two and ten threads, as they induce the highest throughput.

Results. In Figure 4.13, we show the execution breakdown in μ-ops for the RO benchmark. RDMA UpPar requires up to twice more μ-ops to execute the RO benchmark than Slash. Its senders incur in up to 7x more front-end stalls than the senders of Slash. In fact, the execution of its senders with two and ten threads are front-end bound (22 and 33% of total CPU cycles, respectively). In contrast, Slash's senders are essentially core-bound and its receivers are memory-bound.

Discussion. This experiment reveals the source of inefficiency of RDMA UpPar as we compare it to Slash. The key finding is that RDMA UpPar inefficiently use CPU resources due to more complex application logic. This has a two-fold impact on the execution. First, the complex logic behind partitioning results in a large code footprint and thus high number of μ-ops to retire, compared to Slash. A large code footprint results in front-end stalls that slow down the sender and in turn the receiver needs to spend more time waiting. Furthermore, the implementation of partitioning requires branches, which lead to front-end stalls in the case of branch mis-prediction.

Second, the receivers of RDMA UpPar need to poll on multiple RDMA channels (and thus memory locations) depending on fan-out, which makes execution mainly core-bound. Core-bound execution is induced by the *pause* instruction [106] that RDMA channels use for polling (see Section 4.5). This occurs when an RDMA buffer is not fully transferred from remote to local

memory. In contrast, Slash's senders are core-bound, as they saturate the network and thus must wait for data transmission through the *pause* instruction [106]. Its receivers are primarily memory-bound, which is the result of waiting for in-flight data to materialize in registers. Its receivers are also core-bound, as they must wait on senders. However, this differs from the execution of RDMA UpPar. In fact, the senders of Slash cannot send as the network is saturated, whereas the senders of Slash cannot send due to stalls. In sum, the discussion above suggests that the main bottleneck of Slash is the network, in line with our findings of Section 4.7.3.2.

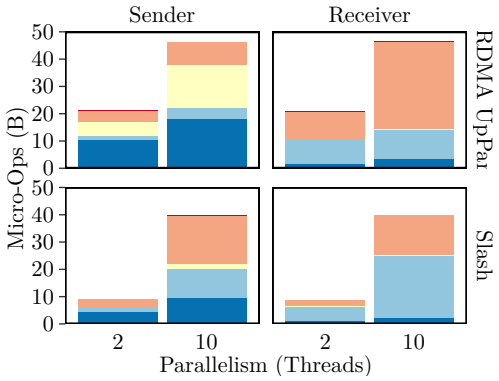


Figure 4.13: Execution Breakdown of RO.

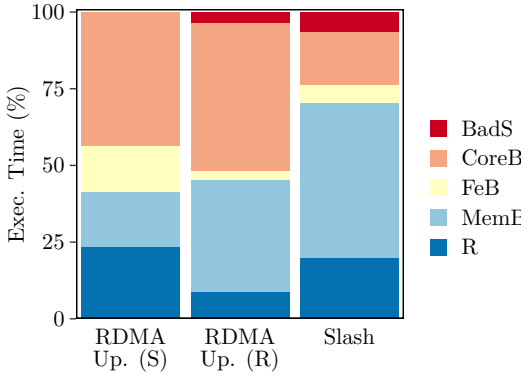


Figure 4.14: Execution Breakdown of YSB.

4.7.3.4 Resource Utilization of Stateful Execution

In the following, we further analyze the usage of CPU resources of Slash and RDMA UpPar to understand the impact of our design choices on stateful queries. We use the best performing configuration as in Section 4.7.3.3 to conduct a micro-architecture analysis of YSB on Slash and the sender and receiver of RDMA UpPar. We collect execution-related hardware performance counters to analyze the following three aspects.

Micro-architectural Analysis. In Figure 4.14, we consider the resource utilization of the CPU micro-architecture and observe that Slash is primarily memory-bound, whereas sender and receiver of RDMA UpPar are core-bound. RDMA UpPar's sender suffers from front-end stalls, as shown by prior findings [47], whereas Slash minimally suffers from branch mis-prediction. Finally, we note that Slash spends about 20% of its execution time performing retirement, whereas the receiver of RDMA UpPar - which computes results - spends 10% of its time retiring instructions.

The reason behind the above observations are as follows. On the sender side of RDMA UpPar, the partitioning logic results in front-end stalls. Besides, the data-dependent writes to fan-out RDMA buffers result in back-end stalls (memory-bound fraction). In total, this accounts for about 30% of the execution time, which slows down partitioning. Receiver threads of RDMA UpPar poll on multiple, inbound RDMA channels, which rely on the *pause* instruction, thereby execution becomes core-bound and slows down the receivers. Note that the sender adjusts its speed to the processing rate of the receiver, which results in waiting that makes the sender core-bound. In contrast, Slash shows a more efficient execution than RDMA UpPar, as it essentially

performs RMWs to the in-memory area of the SSB. RMWs induce a mainly memory-bound execution due to the latency of atomic instructions, such as `compare-and-swap`. The epoch-based state synchronization of Slash results in streamlined RDMA accesses that negligibly impact performance.

Instruction Stream Analysis. We show in Table 4.1 that Slash requires up to 4x less instructions and up to 5x less cycles to process each single record compared to RDMA UpPar. Furthermore, Slash executes close to one *instruction per cycle* (IPC). RDMA UpPar requires up to 0.4 and 0.6 IPC on sender and receiver, respectively. Note that an optimal execution retires 4 instructions per cycle [156].

The difference between the two SUTs lays in the more complex partitioning logic of RDMA UpPar, which needs more instructions per record. Thus, the limiting factor for RDMA UpPar is partitioning, which slows down the receiver. As a result, the consumer essentially waits for inbound data to process, and is thus core-bound. In contrast, Slash executes a simple processing logic on a record basis, yet relies on a more complex logic upon state synchronization. With this trade-off, Slash attains fast execution on the common code path and induces negligible overhead upon synchronization.

Data Locality Analysis. We observe in the rightmost part of Table 4.1 that the execution of Slash induces about 1.5 misses per record on each cache level. In contrast, the producer of RDMA UpPar exhibits about 1.3 misses on each cache level, whereas its receiver minimally suffers from LLC misses. Additionally, Slash induces an aggregated memory throughput of 70.2 GB/s, which is about 52% of the aggregated memory bandwidth of the two nodes. In contrast, RDMA UpPar has a memory access rate of 4.1 (sender) and 4.2 (receiver) GB/s. The LLC misses for RDMA UpPar's sender are due to data dependent writes in RDMA fan-out buffers. Slash is affected by cache misses due to the updates of the SSB, which rely on atomic operations. Partitioning throughput induces a low memory access rate for RDMA UpPar, whereas Slash is mainly memory-bound.

Discussion. This experiment sheds light on the different performance of both Slash SUTs when executing stateful computations. This is due to the more efficient resource utilization of the SSB of Slash versus the data-partitioning approach of RDMA UpPar. In sum, RDMA UpPar is bound by partitioning throughput and ultimately by network bandwidth, whereas Slash is primarily limited by memory performance. This validates that our processing model based on eager computation of partial results and on their lazy merging is an alternative strategy to data re-partitioning of state-of-the-art SPEs.

4.7.4 Summary

In sum, the findings of our experiments validate our design choices. Based on them, we derive the following guidelines for system builders who seek to accelerate their stream processing workloads via RDMA.

1. Apply native RDMA acceleration. Native RDMA acceleration enables a system design that scales with the number of nodes and achieves higher throughput on common streaming

	IPC	Instr./ Rec.	Cyc./ Rec.	L1d Miss/ Rec.	L2d Miss/ Rec.	LLC Miss/ Rec.	Aggr. Mem. Bw (GB/s)
RDMA	0.6	166	274	1.36	1.31	1.2	4.1
UpPar	0.4	78	276	1.74	1.42	0.4	4.2
Slash	0.9	42	53	1.75	1.52	1.3	70.2

Table 4.1: Resource utilization of RDMA UpPar (sender and receiver) and Slash on YSB using two nodes.

workloads than partitioning-based approaches. In particular, our Slash prototype achieves up to 22x and 8x higher throughput than the strongest scale-out baseline on windowed aggregations and joins, respectively. Furthermore, Slash outperforms a state-of-the-art scale-up SPE called LightSaber by a factor of 11.6 on windowed aggregations.

2. Avoid data re-partitioning. Data re-partitioning induces a performance regression, as it makes SPEs be limited by partitioning throughput. To understand this performance regression, we run a performance drill-down of Slash and RDMA UpPar. We show that Slash achieves full line rate on RO benchmark using RDMA and minimal CPU resources. In contrast, RDMA UpPar needs a higher degree of parallelism to reach high throughput, as it is primarily CPU bound due to costly data re-partitioning. Furthermore, we demonstrate that Slash does not suffer from skewed data distributions.

3. Use lazy merging. We show that a processing model based on lazy merging of eagerly computed partial results attains the highest throughput in our evaluation. However, lazy merging requires careful synchronization among nodes to avoid overhead and inconsistency. We demonstrate that our SSB achieves lazy merging, is skew-agnostic, and induces minimum overhead on query processing.

4.8 Related Work

In the following, we discuss the application of RDMA to database systems and the differences between our approach and existing SPEs.

RDMA for database systems. The database community has adopted RDMA to speed up OLAP and OLTP workloads. We identify three areas of adoption: distributed transactions, batch analytical query processing, and key-value stores. Distributed transactions techniques [157] profit from RDMA to scale out on large deployments [127]. Systems that use RDMA for OLTP are Oracle RAC [158], IBM pureScale [159], NAMDB [28, 127], and FaRM [54, 160]. OLAP operators, especially joins, benefit from RDMA to speed up partitioning [28, 130, 161, 162, 131]. Big data frameworks accelerate batch workloads via RDMA [163, 164, 165], while key-value stores use RDMA to increase throughput and reduce latency of value access [166, 129, 167].

Our work is orthogonal to above research, as stream processing has different requirements than traditional database systems or key-value stores [168, 169]. SPEs require fast, stateful processing of inbound data streams. Their key requirement deals with the state management component that must support fast, concurrent point updates and range scans for analytics readiness over

windows. An RDMA-based key-value store, such as FaRM, is not suitable to store state, as it targets transactional workloads with point lookups and updates. As a result, system designers need to devise algorithms and protocols to natively accelerate an SPE using RDMA. With Slash, we fill this gap as we provide SPE components that address RDMA acceleration by design.

Stream processing engines. We distinguish two classes of SPEs. Scale-out SPEs focus on scalability and rely on socket-based communication to distribute query execution on a large cluster of nodes [35, 36, 22, 34, 21]. A recent prototype provides lightweight RDMA integration for Apache Storm [170] and is thus equivalent to RDMA UpPar. Scale-up SPEs target single-node performance but neglect network-related aspects [38, 5, 6, 40, 47]. With Slash, we combine the best of both worlds. Our goal is scale-out stream processing using RDMA acceleration, while considering recent scale-up techniques for SPEs to achieve maximum performance and at rack-scale. To this end, we propose a system design that profits from scale-up techniques, such as omitting partitioning, late merge, shared mutable state, and a hardware-conscious execution. However, we rethink the above techniques to apply them to distributed stateful computation using RDMA and scale processing over multiple nodes. Thus, we devise the RDMA acceleration of late merging and enable consistent shared mutable state among the nodes. Overall, our approach attains higher performance than RDMA-based data re-partitioning approaches, such as RDMA UpPar, and almost linear weak scaling in comparison to a scale-up SPEs, such as LightSaber.

4.9 Conclusion

In this Chapter, we have proposed Slash, our novel SPE with native RDMA acceleration, which paves the way to a new class of SPEs for high-speed networks. Slash enables a processing model based on eager computation of distributed, partial state and its lazy merging into a consistent global state. To this end, the system design of Slash consists of three building blocks that enable stateful processing at full RDMA network speed. We validate our prototype against RDMA UpPar that uses RDMA-based data re-partitioning, IPoIB-enabled Apache Flink, and a scale-up SPE called LightSaber [5]. Overall, we show that our approach achieves on common streaming workloads higher throughput than RDMA UpPar (up to a factor of 22), than Apache Flink (up to two orders of magnitude), and than LightSaber (up to a factor of 11.6). Using the contributions of this Chapter along with the contributions of the previous Chapter, system builders can now accelerate their stateful streaming workloads exploiting the processing capabilities of modern CPUs as well as the increased bandwidths of modern networking technology.

5

Efficient Management of Very Large Distributed State for Scale-out Stream Processing Engines

5.1 Introduction

In the previous Chapters, we provide system designs to achieve efficient stream processing performance on modern compute and networking hardware. However, stream processing requires state management for operators, which requires carefully designed techniques to cope with the tight latency requirements of streaming workload. To this end, in this Chapter, we focus on the management of very large state that SPEs require when reconfiguring long-standing running queries, regardless of the underlying hardware technology.

We observe that increasingly complex analytical queries on real-time data have made Stream Processing Engines (SPEs) over the past years an important component in the big data toolchain. SPEs power stateful analytics behind popular multimedia services, online marketplaces, cloud providers, and mobile games. These services deploy SPEs to implement a wide range of use-cases, e.g., fraud detection, content recommendation, and user profiling [7, 8, 9, 1, 10, 11, 12]. Furthermore, cloud providers offer fully-managed SPEs to customers, which hide operational details [14, 15]. State in these applications scales with the number of users, events, and queries and can reach terabyte sizes [8]. These state sizes originate from intermediate results of large temporal aggregations or joins. We consider state as very large when it exceeds the aggregated main-memory available to the SPE.

To run applications, SPEs have to support continuous stateful stream processing under diverse conditions, e.g., fluctuating data rates and low latency. To handle varying data rates and volumes, modern SPEs scale out stateful query processing [171]. Furthermore, SPEs have to

transparently handle faults and adjust their processing capabilities, regardless of failures or data rates fluctuatations. To this end, they offer runtime optimizations for running query execution plans (QEPs), resource elasticity, and fault tolerance through QEP reconfigurations [1, 34, 172, 173, 12, 171, 113, 2, 174]. State management is necessary to enable fault-tolerance, operator rescaling, and query re-optimization, e.g., load balancing. Therefore, scale-out SPEs require efficient state management and on-the-fly reconfiguration of running queries to quickly react to spikes in the data rate or failures.

The reconfiguration of running stateful queries in the presence of very large operator state brings a multifaceted challenge. First, *network overhead*: a reconfiguration involves state migration between workers over a network, which results in more resource utilization and latency proportional to state size. As a result, this migration overhead increases the end-to-end latency of query processing. Second, *consistency*: a reconfiguration has to guarantee exactly-once processing semantics through consistent state management and record routing. A reconfiguration must thus alter a running QEP without affecting result correctness. Third, *processing overhead*: a reconfiguration must have minimal impact on performance of query processing. An SPE must continuously and robustly process stream records as if no reconfiguration ever occurred.

Today, several industrial and research solutions provide state migration. However, these solutions restrict their scope to small state sizes or offer limited on-the-fly reconfigurations. Apache Flink [175, 1], Apache Spark [36, 2], and Apache Samza [12], enable consistency but at the expense of performance and network throughput. They support large, consistent operator state but they restart a running query for reconfiguration [1, 12, 2]. Research prototypes, e.g., Chi [173], ChronoStream [113], FUGU [172], Megaphone [48], SDG [73], and SEEP [34] address consistency and performance but not network overhead. They enable fine-grained reconfiguration but support smaller state sizes (i.e., tens of gigabytes).

In this Chapter, we show that representatives of the above systems, i.e., Flink and Megaphone, do not cope well with large state sizes and QEP reconfigurations. Although the above systems support stateful processing, they fall short in providing efficient large state migration to enable on-the-fly QEP reconfigurations.

To bridge the gap between stateful stream processing and operational efficiency via on-the-fly QEP reconfigurations and state migration, we propose *Rhino*. Rhino is a library for efficient management of very large distributed state compatible with SPEs based on the streaming dataflow paradigm [15].

Rhino enables on-the-fly reconfiguration of a running query to provide resource elasticity, fault tolerance, and runtime query optimizations (e.g., load balancing) in the presence of very large distributed state. To the best of our knowledge, Rhino is the first system to specifically address migration of large state. Although state-of-the-art systems provide fine-grained state migration, Rhino is optimized for reconfiguration of running queries that maintain large operator state. In particular, Rhino proactively migrates state so that a potential reconfiguration requires minimal state transfer. Rhino applies a state-centric, proactive replication protocol to asynchronously replicate the state of a running operator on a set of SPE workers through incremental checkpoints.

Furthermore, Rhino applies a handover protocol that smoothly migrates processing and state of a running operator among workers. This does not halt query execution and guarantees exactly-once processing. In contrast to state-of-the-art SPEs, our protocols are tailored for resource elasticity, fault tolerance, and runtime query optimizations.

In our evaluation, Rhino reduces reconfiguration time due to state migration by 50x compared to Flink and 15x compared to Megaphone, as shown in Figure 5.1. Furthermore, Rhino shows a reduction in processing latency by three orders of magnitude for a reconfiguration with large state migration. We show that Rhino does not introduce overhead on query processing, even when state is small. Finally, we show that Rhino can handle state migration in a multi-query scenario with a reduction in reconfiguration time by one order of magnitude. Overall, Rhino solves the multifaceted challenge of on-the-fly reconfiguration involving large (and small) stateful operators.

In this Chapter, we make the following contributions:

- We introduce the full system design of Rhino and the rationale behind our architectural choices.
- We devise two protocols for proactive large state migration and on-the-fly reconfiguration of running queries. We design these protocols to specifically handle migrations of large operator state. To this end, Rhino proactively migrates state to reduce the amount of state to move upon a reconfiguration.
- With Rhino, we enable resource elasticity, fault tolerance, and runtime re-optimizations in the presence of very large distributed state. Rhino's state migration is tailored to support these three features.
- Using the NEXMark benchmark suite [149] as representative workload, we validate our system design at terabyte scale against state-of-the-art SPEs.

We structure the remainder of this Chapter as follows. We describe the system design of Rhino in Section 5.2 and its replication and handover protocols in Section 4. We evaluate Rhino in Section 5 and provide an overview of related work in Section 6. Finally, we summarize our work in Section 7.

5.2 System Design

In this section, we present the design of *Rhino*, a library that integrates stateful stream processing with on-the-fly query reconfiguration. First, we present a benchmark that shows the shortcomings of current state management techniques for migration of large operator states (see Section 5.2.1). Second, driven by our findings, we propose an architectural revision for scale-out SPEs to handle large state in Section 5.2.2. Finally, we provide an overview of the protocols and components that we use to implement this architectural change and build Rhino (see Section 5.2.3 to Section 5.2.5).

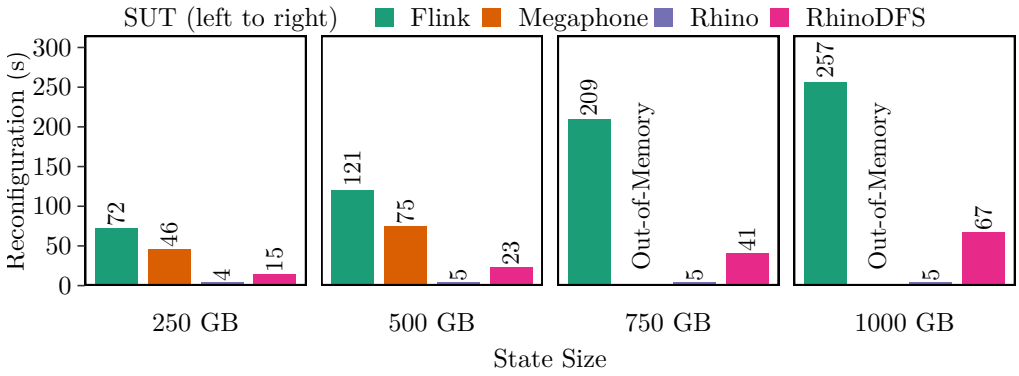


Figure 5.1: Time spent to reconfigure the execution of NBQ8.

5.2.1 Benchmarking state migration techniques

With large state sizes in mind, we run a benchmark to assess the stateful capabilities of modern scale-out SPEs. We select Flink as a representative, industrial-grade SPE due to its wide adoption and its built-in support for state. We assume a distributed file system (DFS) [176, 177, 178] to be commonly deployed with scale-out SPEs for checkpoint storage [1, 12]. Furthermore, we choose Megaphone [48] as it is the most recently proposed scale-out state migration technique. Chi [173] is another SPE that provides state migration, yet it is not available for public access at the time of writing.

In Figure 5.1, we report the impact on processing latency during a reconfiguration with varying state size (i.e., ranging from 250 GB to 1 TB) and show results in Figure 5.1. We observe that Flink requires a full query restart, which results in a significant latency spike that hinders processing performance. Our time breakdown, which we fully describe in Section 5.4.2.1, indicates that Flink spends the majority of the time in state materialization from a previous checkpoint prior to resuming query processing. Upon a reconfiguration, a parallel instance of a stateful operator retrieves in bulk its new state from DFS. Every instance quickly retrieves its local blocks (if any), yet the materialization of remote blocks entails network transfer. As a result, state retrieval introduces additional latency because every instance pulls its state from multiple workers.

Although Megaphone provides fine-grained state migration, we find that it does not handle large state size. All its executions in our benchmark with more than 500 GB of state terminated with an out-of-memory error. By inspecting its source code [179], we consider the lack of memory management to support state migration as the main responsible for this error. However, we observe that Megaphone can successfully complete fine-grained state migrations, if state fits main memory. Note that the introduction of a data structure that supports out-of-core storage, e.g., a key value store (KVS), would enable Megaphone to store larger-than-memory state. However, this does not solve the problem of large state migration as Megaphone migrates key-value pairs in one batch.

5.2.2 Overview of Rhino

In the previous section, we show that current SPEs do not properly support large operator state when reconfiguring a running query. To fill this gap, we build Rhino to support large state and enable on-the-fly reconfiguration of running queries by design. Rhino is thus tailored to provide fine-grained fault-tolerance, resource elasticity, and runtime optimizations, e.g., load balancing.

The design of Rhino focuses on efficient reconfigurations and migration of large state. To this end, Rhino introduces three key techniques. First, Rhino proactively migrates state such that a reconfiguration requires minimal state transfer. In contrast to the state-of-the-art, Rhino proactively and periodically persists the state of an operator instance on exactly r workers of the SPE. Rhino thus executes fast reconfiguration (handover) from an origin instance to a target instance that runs on one of the r workers. Second, to control the state size to be migrated, Rhino asynchronously replicates incremental checkpoints of the state. As a result, Rhino migrates only the last incremental checkpoint of the state of an operator upon a handover. In contrast, other systems transfer state in bulk [1, 48]. Third, Rhino introduces consistent hashing with virtual nodes to further partition the key partition of an operator instance. In Rhino, virtual nodes are the finest granularity of a reconfiguration.

5.2.3 Components Overview

Rhino introduces four components in an SPE, i.e., a *Replication Manager (RM)*, a *Handover Manager (HM)*, a distributed runtime, and modifications to existing components.

Replication Manager. The replication manager runs on the coordinator of the SPE and builds the replica groups of each instance based on the bin-packing algorithm. After that, the RM assigns replicas to workers. We implement this component with less that 1K lines of code (loc).

Handover Manager. The handover manager serves as the coordinator for in-flight reconfiguration with state transfer. Based on a human or automatic decision-maker (e.g., Dhalion [180], DS2 [181]), our HM starts a reconfiguration. After that, it monitors Rhino's distributed runtime (see below) for a successful and timely completion of the triggered handover. We implement this component in about 1K loc.

Distributed Runtime. The distributed runtime runs on the workers of the SPE. This runtime uses our two application-level protocols for handover and state-centric replication. We implement this runtime with less than 5K loc.

Modifications. Besides introducing new components, Rhino also extends stateful operators to 1) receive and broadcast control events on their data channel, 2) buffer in-flight record of specific data channels, 3) reconfigure inbound and outbound data channels, and 4) ingest checkpointed state for one or more virtual nodes of an instance. In total, these modifications require about 2K loc in Flink.

5.2.4 Host System Requirements

We design Rhino as a library that can be deployed on top of a scale-out SPE. However, the target SPE has to fulfill the following requirements.

R1: Streaming dataflow paradigm. Our handover protocol requires the SPE to follow the streaming dataflow paradigm because Rhino relies on markers that flow along with records from source operators. Common SPEs feature this paradigm with two processing models: the record-at-a-time model, adopted by SPEs such as Apache Flink and Apache Samza [1, 12], and the bulk synchronous processing model, adopted by SPEs such as Apache Spark Streaming [2]. With a record-at-a-time model, Rhino can inject a marker in the record flow at any point in time. In contrast, the bulk synchronous processing model introduces coarse-grained synchronization at the end of every micro-batch. Rhino could piggyback a handover on this synchronization barrier to enable reconfigurations at the expense of increased latency.

R2: Consistent hashing with virtual nodes. Our handover protocol requires techniques for fine-grained reconfiguration. A fine-grained reconfiguration requires fine-grained state migration and record rerouting. To this end, Rhino uses consistent hashing with virtual nodes [182] instead of traditional consistent hashing [183, 184]. This enables Rhino to reassign state to instances and route records at a finer granularity, upon a reconfiguration. Thus, Rhino further partitions streams and state into a fixed number of virtual nodes, which are the finest granularity of a reconfiguration.

R3: Mutable State. It is necessary for Rhino that each operator supports local, mutable state along with distributed checkpointing [1, 137]. In addition, Rhino requires read and write access to the internal state of a parallel instance to update it upon a handover. Embedded KVS, e.g., RocksDB [185] and FASTER [134], provide mutable state compatible with Rhino. We do not consider external KVS, e.g., Cassandra [182] and Anna [139], as they would introduce extra synchronization in a distributed environment.

To build our system prototype, we select Apache Flink as host system to deploy Rhino as it meets R1 and R3 (via RocksDB) by design and requires minimal changes to meet R2. However, Rhino is compatible with any SPE that fulfills the above requirements, e.g., Apache Spark, Apache Storm, and Google Dataflow.

5.2.5 Benefits of Rhino

In this section, we show how current SPEs could benefit from Rhino and its ability to reconfigure running queries at a fine granularity and in different scenarios. In particular, Rhino enables higher operational efficiency through fine-grained load-balancing (see Section 5.2.5.1), resource elasticity (see Section 5.2.5.2), and fault-tolerance (see Section 5.2.5.3). Note that the handover protocol ensures that query execution preserves exactly-once processing semantics during all operations.

5.2.5.1 Load balancing

Rhino enables fine-grained load balancing, which is missing in current SPEs [186, 48]. Load balancing is beneficial when the SPE detects that a physical instance O is overwhelmed (e.g., due to data skewness) but instance T is underloaded. In this case, the SPE requests Rhino's HM to migrate the processing and state of some virtual nodes of O to T, which runs on a worker that has a copy of the state of O. After that, the HM triggers a handover that involves these virtual nodes. When the handover completes, T takes over the processing of migrated virtual nodes of O.

5.2.5.2 Resource Elasticity

Rhino enables resource elasticity for current SPEs and handles rescaling as a special case of load balancing. To this end, Rhino moves some virtual nodes of a running instance to a newly spawned instance. In particular, Rhino enables adding operator instances on an in-use worker (i.e., vertical scaling) and deploys more instances on newly provisioned workers (i.e., horizontal scaling).

Overall, Rhino follows a similar course of action for resource elasticity as for load balancing. Rhino defines O and T as the origin and target instances, respectively. In the case of vertical scaling, Rhino assumes that the SPE deploys T on an in-use worker, which has a copy of the state of O. In the case of horizontal scaling, Rhino provisions a new worker, which requires a bulk copy of the state. The cost of a bulk transfer is mitigated by early provision of workers and parallel copies.

5.2.5.3 Fault Tolerance

Rhino enables fine-grained fault tolerance that results in faster recovery time (see Figure 5.1). Upon the failure of O, the SPE requests the HM to migrate O on a the workers that stores a copy of its state and can run a new instance T. Thus, Rhino triggers a handover that instructs T to start processing using the last checkpointed state of O. In addition, the handover instructs upstream and downstream operators to rewire their channels to connect to T.

5.3 The Protocols

In this section, we describe Rhino's protocols in detail. We first introduce the handover protocol in Section 5.3.1 and then the replication protocol in Section 5.3.2.

5.3.1 Handover Protocol

In this section, we define our handover protocol (see Section 5.3.1.1), its steps (see Section 5.3.1.2), and correctness (see Section 5.3.1.3).

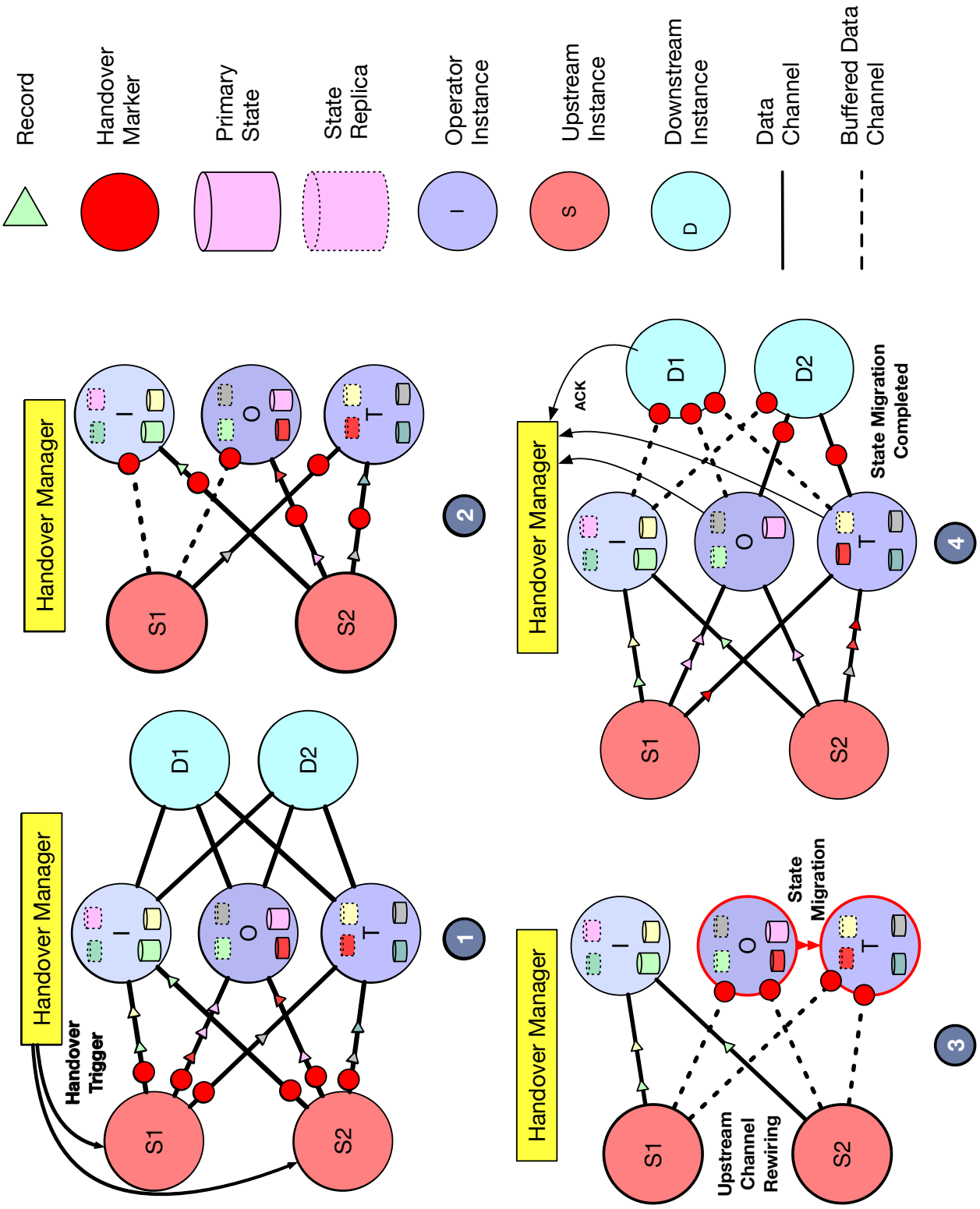


Figure 5.2: Steps of the Handover Protocol of Rhino.

5.3.1.1 Protocol Description

To enable fine-grained reconfiguration of a running query, our handover protocol defines three aspects. First, it defines the concept of configuration epochs for a running stateful query. Second, it defines how to transition from one configuration epoch to the next. Third, it defines what properties hold during a handover.

Configuration epoch. Our protocol discretizes the execution of a query into configuration epochs of variable length. A configuration epoch $E_{a,b}$ is the non-overlapping time interval betweens two consecutive reconfigurations that occur at time a and b, respectively. An epoch is independent from any windowing semantics or checkpoint epochs. The first epoch begins with the deployment of a query. During an epoch, each instance processes records using fixed parameters, i.e., assigned worker, input and output channels, state partition, and assigned virtual nodes. The task of the handover is to consistently reconfigure the parameters of a running instance. This involves state migration as well as channel rewiring. Furthermore, it triggers the transition to the next epoch.

Handover. A handover reconfigures a non-source operator of a running query at time t. This ends the epoch $E_{a,t}$ and starts $E_{t,b}$. As a result, records with timestamp in interval $[a, t]$ are processed with a configuration and records with timestamp in $[t, b]$ are processed with a new configuration.

A handover involves 1) the propagation of handover markers, 2) state migration, and 3) channel rewiring. A handover marker (1) is a control event that flows from source operators to the instances to reconfigure through all dataflow channels. A handover marker carries the configuration update that involves the origin and target instances and their upstream and downstream instances. This is inspired by Chi's idea of embedding control messages in the dataflow to reconfigure queries [173]. As soon as the origin instance receives a marker on all its inbound channels, it migrates state (2) to the target instance. When the target instance receives its markers and the state from the origin instance, it takes over processing. Channel rewiring (3) ensures that records are consistently routed among upstream, downstream, origin, and target instances.

Epoch alignment. To perform a consistent reconfiguration at time t, we use handover markers to induce an instance-local synchronization point. To this end, we use an epoch alignment algorithm, similarly to the checkpointing approach of Carbone et al. [1]. Recall that an instance nondetermistically polls records and control events from its channels. Therefore, it is necessary to ensure a total order among otherwise partially ordered records. When a marker with timestamp t arrives at an instance from a channel, it signals that no record older than t is to be expected from that channel. Since newer records have to be processed with the new configuration, the instance has to buffer upcoming records on that channel to process them later. When an instance receives all markers, no record older than t is in-flight. As a result, the instance reaches its synchronization point and the reconfiguration takes place. Target instances, however, must wait for state migration to complete.

5.3.1.2 Protocol Steps

In this section, we define the steps of our handover protocol. To do so, we first provide an example and then a formal description of each step.

Example. The SPE requests Rhino to migrate the processing of red records from origin instance O to target instance T (see Figure 5.2). Instances O and T run on distinct workers. To fulfill the SPE's request, the HM triggers a handover and source instances S1 and S2 inject a handover marker into the dataflow (1). When receiving a marker on a channel, instances O and T buffer records further arriving on that channel (2). Upon receiving all markers, O migrates the red state (3). At the completion of the state transfer, O and T acknowledge the HM (4). Next, upstream instances S1 and S2 send red records to T instead of O. Finally, instance O, T, D1, and D2 acknowledge the HM.

Step 1. Assume that running instances completed a checkpoint at time t'. Based on a policy, an SPE triggers a handover at time t between the origin instance O and the target instance T for a virtual node $(k_l, k_q]$. The handover assumes previously checkpointed state of O to be replicated on the worker running T and that no checkpoint is in-flight.

Step 2. The protocol defines the following actions for an operator instance, which we summarize in Figure 5.2.

(1) Source operators injects a handover marker h_t in their outbound channels.

(2) When an instance I receives h_t on one of its inbound channel, it buffers incoming records on that channel.

(3) Upon receiving h_t on all its inbound channel, I broadcasts a handover marker on its outbound channels and performs Step 3.

(4) When T receives the checkpointed state for $(k_l, k_q]$, it processes buffered records that arrived after m_t.

Step 3. According to its position (upstream or downstream) with O and T, I reacts using one of the following routines. First, if I is an *upstream* instance, it rewires the output channels for $(k_l, k_q]$ to send records to T. Second, if I is a *downstream* instance and T is a new instance, it rewires its inbound channels to process records from T. Third, if I is the origin instance, it triggers a local checkpoint t, which Rhino transfers to T, and releases unneeded resources. Fourth, if I is the target instance, it loads checkpointed state of O and then consumes buffered records. If origin instance has failed, Rhino does not migrate state and relies on upstream backup to replay records, if necessary. Note that operators are aware of an in-flight handover and ignore seen records based on their timestamps. As a result, O processes records of $(k_l, k_q]$ with a timestamp smaller than t, whereas T processes records with a timestamp greater than t.

Step 4. As soon as an instance completes its steps, it acknowledges the HM. When all instances successfully send an acknowledgment, the HM marks the handover as completed.

Our handover protocol needs upstream backup when a failure occurs. The upstream backup can be a source operator or windowed operator that keeps in-flight windows. They can replay

records from an external system or the window state. In the latter case, downstream operators must acknowledge when it is safe to delete the content of a window.

Note that the handover protocol is not fault-tolerant. A failure that occurs during a handover may restart the protocol. We plan to make our protocol fault-tolerant as future work, e.g., via fine-grained, upstream buffering of records.

5.3.1.3 Correctness Guarantees

A handover between two instances has the correctness guarantees defined by Theorem 1.

Theorem 1 *Consider a handover that migrates the state S^{t-1} of the virtual node $(k_l, k_q]$ from O to T at timestamp t. The protocol guarantees that: 1) T receives S^{t-1} at t and then processes records with keys in $(k_l, k_q]$ and timestamps greater than t and 2) the handover completes in finite time.*

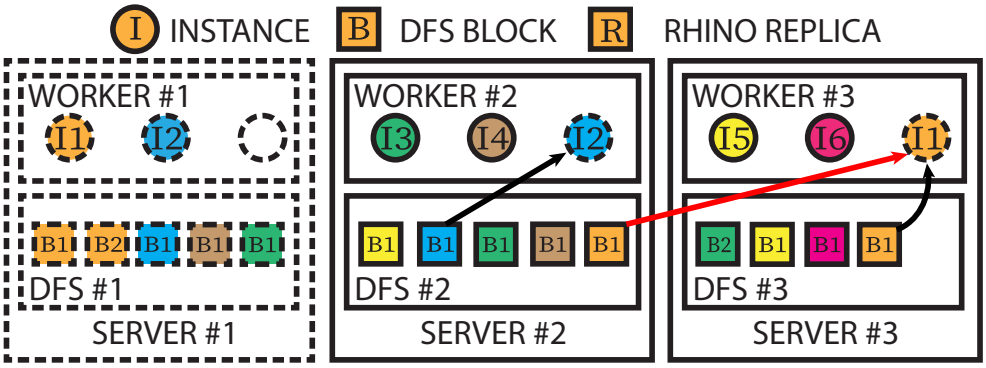
Proof. Let $t-1$ be the timestamp of the last processed record and S^{t-1} the state of O and assume that T has received all previous incremental checkpoints from O. Recall that an instance is aligned at time t if it has processed all records bearing timestamps smaller than t. In our setting, the alignment at time t occurs when an instance receives the handover marker h_t on its channels and has no in-flight checkpoint. The alignment at an upstream instance results in the routing to T of records with keys in $(k_l, k_q]$ and timestamp greater than t. Note that these records are not processed until the handover on T occurs. The alignment at t on O results in a incremental checkpoint of the state S^{t-1}, which is migrated to T. The alignment of T occurs upon receiving markers on its channels and the incremental checkpoint of S^{t-1} from O. Thus, T receives S^{t-1} at time t and only then processes records with keys in $(k_l, k_q]$ and timestamps greater than t (1). State transfer, channel rewiring, and acknowledgment to coordinator are deterministic operations. In addition, channels are FIFO and durable, thus, they eventually deliver handover markers to instances. As a result, a handover completes in finite time (2).

5.3.2 Replication Protocol

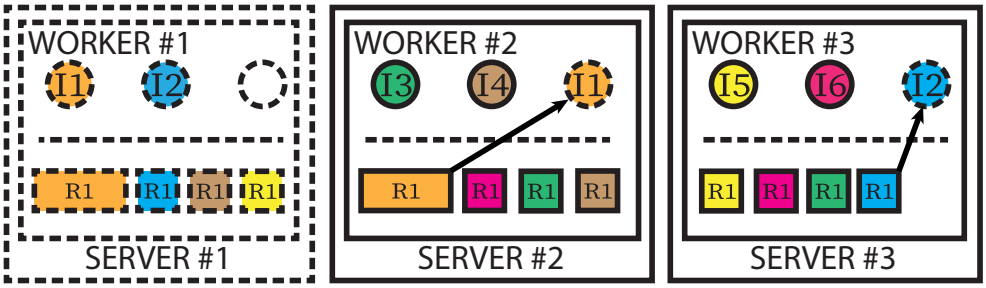
In this section, we describe our state replication protocol and how we use it to guarantee timely handovers between two stateful instances.

5.3.2.1 Protocol Description

Our state-centric replication protocol enables us to asynchronously replicate the local checkpointed state (primary copy) of a parallel instance of an operator on r workers of the SPE (secondary copies). It assumes that the SPE periodically triggers global checkpoints. Rhino replaces the *block-centric* replication of traditional DFS with a *state-centric* protocol. The intuition behind our protocol is that Rhino proactively migrates state of each instance through incremental checkpoints to ensure that a target instance during a handover has the latest copy of its state. The protocol assumes each worker to have dedicated storage units to locally fetch



(a) Block-centric.



(b) State-centric.

Figure 5.3: Block-centric vs. state-centric replication. Black and red arrows indicate local and remote fetching, respectively.

checkpointed state. This allows the SPE upon a reconfiguration to spawn or reuse an instance on a worker that already stores the state for this instance. Traditional DFSs and disaggregated storage do not guarantee this property as block placement is transparent to client systems. As a result, the SPE must query the DFS to retrieve necessary blocks. The retrieval is either local or remote depending on the placement of each block.

In Figure 5.3, we show an example of configuration of block-centric and state-centric replication. With block-centric replication, the state of each operator instance is split into multiple blocks, which are replicated on 3 servers. With state-centric replication, the state of each operator instance is entirely replicated on 2 servers. This improves replica fetching upon a recovery. When server 1 fails, its stateful instances are to resume on servers 2 and 3. With block-centric replication, server 3 fetches B1 and B2 from server 2. With state-centric replication, fetching is local.

5.3.2.2 Protocol Phases

Our state-centric replication protocol consists of two phases. The first phase of the protocol instructs how to build *replica groups* for each stateful instance. A *replica group* is a chain of r workers, which owns the secondary copy of the state of a parallel instance, i.e., the state of all virtual nodes. The second phase defines how to perform asynchronous replication of a primary copy to its *replica group*.

Phase 1: Protocol Setup. The RM configures the protocol at deployment-time of a query or upon any change in a running QEP. Through a placement algorithm, such as bin packing, the RM creates a replica group of r distinct workers for each stateful instance. In this part, we assume that workers have equal capacities and use all available workers for the replication of the global state. However, we leave the investigation of further placement algorithms and heuristics as future work. Overall, the RM assigns to every worker a set of secondary copies to maintain.

Phase 2: Distributed Replica Transfer. Our distributed replication runtime implements a state-centric replication protocol that follows the primary/secondary replication pattern [187, 188] with chain replication [189]. We choose chain replication as it guarantees parallel replication with high network throughput. In addition, we use credit-based flow control [143] for application-level congestion control.

Each member of the chain asynchronously stores the received data to disk and forwards the data to its successor in the chain. As soon as the last worker in a chain stores the copy of the checkpointed state to disk, it acknowledges its predecessor in the chain, which does the same. When the owner of the primary copy receives the acknowledgment, it marks the checkpoint as completed and releases unnecessary resources. The replication of a checkpoint completes successfully only if the runtime successfully acknowledges all copies within a timeout. When the replication process is completed, Rhino marks the checkpoint as completed.

5.3.2.3 Correctness Guarantees

Our replication protocol strictly follows the fault-model behind chain replication [189]. We assume *fail-stop* workers that halt upon a failure, which they report to replication manager (RM). Our protocol distinguishes two failures: 1) failure on a running operator and 2) failure on the replica group. If the primary owner fails, the RM halts the replication for the failed operator and only when it resolves the failure, the replication resumes. If a worker in a replica group fails, the RM removes the failed worker and replaces it with another worker.

5.4 Evaluation

In this section, we experimentally validate the system design of Rhino and compare it against three systems under test (SUTs [190]): Apache Flink, Megaphone, and RhinoDFS (a variant that uses HDFS for state migration). First, we describe the setup of our evaluation in Section 5.4.1. After that, we evaluate state migration in the case of a virtual machine (VM) failure in Section 5.4.2. In Section 5.4.3, we show the overhead that Rhino introduces during query processing. Next, we evaluate vertical scaling (see Section 5.4.4.1) and load balancing (see Section 5.4.4.2). Finally, we examine our SUTs under varying data rates (see Section 5.4.5) and sum up the evaluation (see Section 5.4.6).

5.4.1 Experiment Setup

In this section, we introduce our experimental setup (see Section 5.4.1.1), our workloads (see Section 5.4.1.2), SUT configuration (see Section 5.4.1.3), stream generator (see Section 5.4.1.4), and SUT interaction (see Section 5.4.1.5).

5.4.1.1 Hardware and Software

In our experiments, we deploy our SUTs on a mid-sized cluster that we rent on Google Cloud Platform. The cluster consists of 16 `n1-standard-16` virtual machines. Each VM is equipped with a 16-vcores Intel Xeon (Skylake family) running at 2 Ghz, 64 GB of main memory, and two local NVMe SSDs. These VMs run Debian 9 Cloud, use the OpenJDK JVM (version 1.8), and are connected via a 2 Gbps/vcore virtual network switch.

In our evaluation, we use Apache Flink 1.6 as baseline and Apache Kafka 0.11 as a reliable message broker that provides upstream backup [191]. Apache Flink ships with RocksDB [185] as out-of-core storage layer for state. Each stateful operator instance in Apache Flink has its own instance of RocksDB. We pair Flink with the Apache Hadoop Distributed File System (HDFS) version 2.7 as persistent checkpoint storage. Finally, we use the Megaphone implementation (rev: `abadf42`) available online [179].

5.4.1.2 Workloads

We select the NEXMark benchmark suite to experimentally validate our system design [149]. This suite simulates a real-time auction platform with auction, bids, and new user events. The NEXMark benchmark consists of three logical streams, i.e., an auction stream, a bid stream, and a new user event stream. Records are 206 (new user), 269 (auction), and 32 (bid) bytes large. Every record stores an 8-bytes primary key and an 8-bytes creation timestamp. The suite features stateless (e.g., projection and filtering) and stateful (e.g., stream joins) queries.

We use query 5 (NBQ5) and query 8 (NBQ8) and we define a new query (NBQX) that consists of five sub-queries. Based on the these queries, we define three workloads to assess our SUTs. First, NBQ5 contains a window aggregation with a window of 60 secs and 10 secs slide on the bid stream. We select NBQ5 as it features small state sizes and a read-modify-write state update pattern. Second, NBQ8 consists of a tumbling window join on 12 hours in event time on the auction and new user streams. We choose NBQ8 since it reaches large state sizes due to its append state update pattern. Third, NBQX contains multiple four session window joins with 30, 60, 90, 120 minutes gap and a tumbling window join of four hours on the auction and bid stream. We select NBQX as it contains multiple queries that alone have mid-sized state but result in large state size by running concurrently. Furthermore, NBQX features append and deletion update patterns.

5.4.1.3 SUT Configuration

We deploy Flink, Megaphone, RhinoDFS, and Rhino on eight VMs, Kafka on four VMs, and our data generator on four VMs. We use this configuration to ensure that Kafka and our data generator have no throughput limitations. We spawn HDFS instances on the same VMs where the SUTs run. In addition, we reserve one VM to run the coordinators of Flink and Rhino.

Flink and Rhino. To configure Flink (and Rhino), we follow its configuration guidelines [148]. On each VM, we allocate half cores for processing and the other half for network and disk I/O. We allocate 55% of the OS memory to Flink and Rhino, which they divide equally in on-heap and off-heap memory. Flink and Rhino use the off-heap memory for buffer management and on-heap memory for data processing. We assign the remaining memory to RocksDB and the OS page cache. In addition, we assign one dedicated SSD to Rhino's replication runtime. Finally, we use 2^{15} key groups for consistent hashing and 4 virtual nodes for Rhino as these values lead to best performance for our workloads.

Megaphone. We configure Megaphone to use half of the cores for stream processing and the other half for network transfer, following TimelyDataflow's guidelines [179]. In addition, we configure Megaphone to use 2^{15} bins to be compliant to Rhino's setup.

RocksDB. We configure RocksDB for SSD storage, following best practices [192]. To this end, we use fixed-sized memtables (64 MB), bloom filters (for point lookup), and 64 MB as block size for *string sorted tables*.

HDFS. We configure HDFS for Flink and RhinoDFS with a replication factor of two, i.e., each block is replicated to two VMs. As a result, HDFS replicates the first copy of a state block locally and the second block on a different VM. To simulate this replication factor, we configure Rhino to store a local copy (primary copy) and a remote copy (secondary copy) of the state.

Kafka. We let Kafka use all SSDs, all cores, and 55 GB of page cache on each VM, following best practices [193]. We configure Kafka to batch records in buffers of 32 KB with a maximum waiting time of 100 ms. We use three Kafka topics with 32 partitions each to represent 32 new user, 32 bid, and 32 auction streams.

5.4.1.4 Stream Generator

We implement a stream generator to produce three logical streams of new user, bid, and auction events. Our generator concurrently creates 32 physical streams for each logical stream. To minimize JVM overhead, we implement our generator by omitting managed object allocation and garbage collection.

To assess the SUTs, we let our generator produce records at the maximum sustainable throughput of all SUTs. The maximum sustainable throughput is the maximum rate at which an SPE can ingest a stream and output results at constant, target latency [194]. To this end, we experimentally find the sustainable throughput for each query. Thus, we configure the generator to generate each stream at 128 MB/s for NBQ5 (target latency 500 ms) and 8 MB/s for NBQ8 (target latency 500 ms) and NBQX (target latency 5 s). Overall, we deploy four generators with

eight running threads each for a total throughput of 4 GB/s (~135 M records/s) for NBQ5, 256 MB/s for (~500 K records/s) for NBQ8, and 256 MB/s (~3.45 M records/s) for NBQX. Primary keys (auction id, person id) are generated randomly following uniform distribution.

Note that Megaphone does not support real-time data ingestion via external systems as this results in increasing latency. Therefore, Megaphone generates records on-demand within its source operators.

5.4.1.5 SUT Interaction

Unless stated otherwise, we run our SUTs in conjunction with Kafka and our own custom generator. We follow the methodology of Karimov et al. [194] for end-to-end evaluation of SPEs. To this end, we decouple the data generator, Kafka, and SPE. We denote the end-to-end processing latency as the interval between the arrival timestamp at the last operator in a pipeline and the creation timestamp of the record. Our generator creates timestamped records in event time and sends them to Kafka, which acts as reliable message broker and replays streams when necessary.

We implement our queries in Flink and Rhino using built-in operators. Each source runs with a degree of parallelism (DOP) of 32 to fully use Kafka parallelism, i.e., one thread per partition. We run the join and aggregation operators with a DOP of 64 to fully use SUT parallelism. We instrument the join and aggregation operators to measure processing latency. We repeat each experiment multiple times and report mean, minimum, and 99-th percentile measurements.

5.4.2 Rhino for Fault Tolerance

In this set of experiments, we examine two important aspects of an SPE. First, we evaluate the performance that a modern SPE exhibits in restoring a query with large operator state (see Section 5.4.2.1). Second, we measure the impact of recovering from a failure on processing latency (see Section 5.4.2.2).

5.4.2.1 Recovery Time

In this experiment, we run a benchmark using NBQ8 to measure the time required for our SUTs to recover from a failure with varying state sizes. Then, we perform a time breakdown of the time spent in the recovery process. Finally, we compare our results, which we summarize in Table 5.1.

Workload. For this experiment, NBQ8 runs until it reaches the desired state size of 250 GB, 500 GB, 750 GB, and 1 TB. Next, we terminate one VM and measure the time spent by our SUTs to reconfigure the running query. Furthermore, we configure the interval of incremental checkpointing to three minutes for Flink and Rhino variants.

We distinguish three operations that comprise a recovery, i.e., scheduling, state fetching, and state loading. Scheduling is the time spent in triggering a reconfiguration . State fetching is the time spent in retrieving state from the location of the previous checkpoint. State loading is the time spent in loading checkpointed state into the state backend.

State Size	SUT	Scheduling	State Fetching	State Loading
250 GB	Flink	2.2 ± 0.1	68.2 ± 5.7	1.3 ± 0.2
	Rhino	2.8 ± 0.2	0.2 ± 0.1	1.3 ± 0.3
	RhinoDFS	2.9 ± 0.2	10.7 ± 3.1	1.3 ± 0.1
	Megaphone	46.3 ± 2.8		
500 GB	Flink	2.5 ± 0.2	116.6 ± 4.9	1.8 ± 0.3
	Rhino	3.1 ± 0.3	0.2 ± 0.1	1.3 ± 0.3
	RhinoDFS	3.0 ± 0.3	18.9 ± 3.7	1.3 ± 0.5
	Megaphone	74.8 ± 3.0		
750 GB	Flink	2.6 ± 0.3	205.3 ± 5.2	1.3 ± 0.1
	Rhino	3.0 ± 0.2	0.2 ± 0.1	1.5 ± 0.1
	RhinoDFS	2.6 ± 0.1	36.1 ± 2.3	1.5 ± 0.2
	Megaphone	Out-of-Memory		
1000 GB	Flink	2.4 ± 0.3	252.9 ± 5.9	1.5 ± 0.2
	Rhino	3.0 ± 0.2	0.2 ± 0.1	1.5 ± 0.2
	RhinoDFS	2.9 ± 0.3	62.7 ± 0.9	1.5 ± 0.1
	Megaphone	Out-of-Memory		

Table 5.1: Time breakdown in seconds for state migration during a recovery.

Result. The time-breakdown in Table 5.1 shows three aspects. First, we observe that the most expensive operation for block-centric replication is state fetching for RhinoDFS and Flink. This dominates the overall recovery process and depends on the state size. In contrast, Rhino's state-centric replication reduces the state fetching overhead significantly. Performance increases due to local state fetching, which involves hard-linking instead of network transfer.

Second, scheduling and state loading have negligible impact on the recovery duration. In particular, scheduling is about 25% faster in Flink because Flink triggers a recovery immediately. In contrast, Rhino starts a reconfiguration through handovers that reach target instances based on their processing speed. For Rhino and Flink, state loading in RocksDB only needs to open the data files and process metadata files. After the load, the state resides in the last level of the LSM-Tree and is ready to be queried by the SPE.

Third, we observe that Megaphone does not support workloads with more than 500 GB of state as it runs out of memory. For state smaller than 500 GB, Megaphone spends the majority of time to schedule migrations, serialize state into buffers, write buffers on the network, deserialize buffers, and restore state. Note that Megaphone overlaps the above operations on a key basis, e.g., it schedules and migrates state of different keys at the same time. Therefore, it was unfeasible for us to break down its migration times.

Discussion. This experiment outlines that Rhino efficiently recovers a query from a VM failure in a few seconds, even in the presence of large operator state. This improvement is due to local state fetching, which is only marginally impacted by state size. In contrast, state fetching in Flink and RhinoDFS entails network transfer and introduces latency proportional to the state size. In particular, RhinoDFS is faster than Flink because of fine-grained operator restart but

does not achieve low-latency. Instead, Flink fully restarts NBQ8, which results in a 50x higher latency. In our setup, Megaphone does not reach all desired state sizes due to lack of memory management for state-related operations. This prevents Megaphone from supporting large state workloads. In the cases where Megaphone can sustain a workload, its state migration is up to 1.5x faster than Flink's one. Overall, Rhino improves recovery time by 50x compared to Flink, 15x compared to Megaphone, and 11x compared to RhinoDFS.

Note that we run the same benchmark on NBQ5 and all SUTs had comparable recovery time. This suggests that Rhino's design is beneficial if queries require large state.

5.4.2.2 Impact on Latency

In this experiment, we examine the impact of state migration on the end-to-end processing latency in Flink, Rhino, and RhinoDFS. We exclude Megaphone in the following experiments as it provides state migration but does not have mechanisms for fault-tolerance and resource elasticity. Therefore, the optimizations provided by Megaphone are orthogonal to our work.

Workloads. In this experiment, we run NBQ8, NBQ5, and NBQX on eight VMs. We terminate one VM after three checkpoints and let each system recover from the failure. Upon the failure, the overall state size of the last checkpoint before the failure is approximately 190 GB (NBQ8), 26 MB (NBQ5), and 180 GB (NBQX). After the failure, we let the SUTs run for other three checkpoints and then we terminate the execution. In Figure 5.4a, 5.4b, and 5.4c, we report our latency measurements, which we sample every 200 K records.

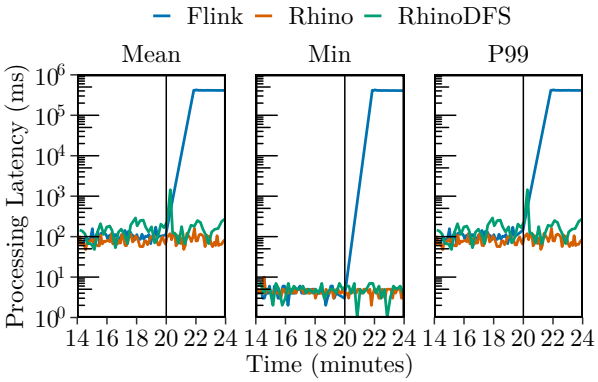
Result. Figures 5.4a, 5.4b, and 5.4c shows that Rhino and Flink can process stream events with low overhead at steady state. For NBQ8 and NBQ5, the average latency for both systems is stable around 100 ms (min: 2 ms, p99: 6.8 s). For NBQX, the average latency is 5 s (min: 1 s, p99: 12 s). Note that the higher max latencies are due to the synchronous phase of a checkpoint, which pauses query processing.

Upon a VM failure, we observe that latency in Rhino is not affected, whereas the latency of Flink increases up to 300 s. After recovery, Rhino's variants do not exhibit any impact on latency. In contrast, Flink accumulates up to 300 s of latency lag from the data generator and cannot resume processing with sub-second latency after an outage.

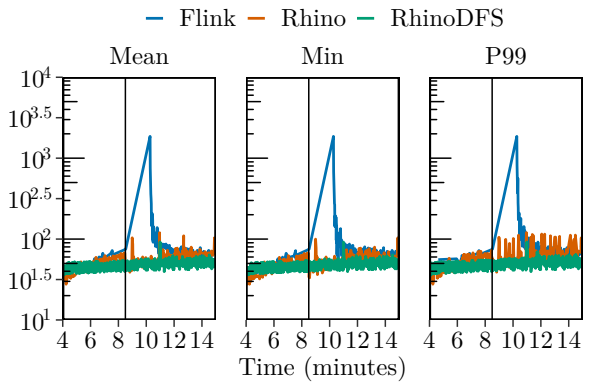
Discussion. Overall, our evaluation confirms that Rhino's design choices result in a robust operational behavior. Rhino handles a VM failure without increasing latency, whereas Flink increases latency by three orders of magnitude.

The main reasons for Flink's decreased performance is three-fold. First, Flink needs to stop and restart the running query, which takes a few seconds. Second, deployed operator instances need to fetch state through HDFS prior to resuming processing. Third, Flink needs to replay records from the upstream backup, which negatively affects latency, whereas Rhino buffers records internally.

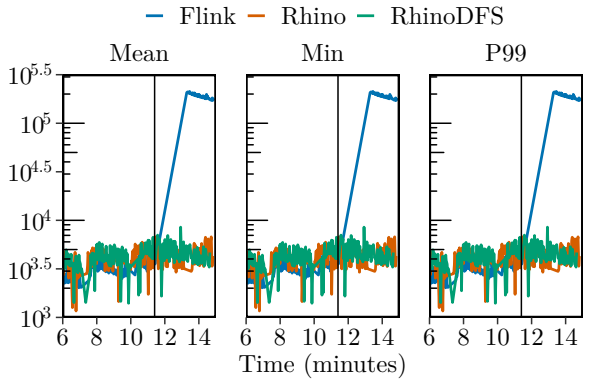
Although latency in Rhino is stable after a handover, the SPE is expected to provision a new worker to replace the failed VM. Afterwards, Rhino can migrate again the operators to rebalance the cluster.



(a) Fault Tolerance NBQ8.



(b) Fault Tolerance NBQ5.



(c) Fault Tolerance NBQX.

Figure 5.4: End-to-end Processing Latency in ms (using log-scale) for our fault tolerance experiments. The vertical black bar represents the moment we trigger a reconfiguration.

5.4.3 Overhead of Rhino

In this experiment, we evaluate the impact of state-centric replication on query processing. To this end, we measure latency and OS resource utilization with the SUTs running below their saturation

point, i.e., the generator rate is set to maximum sustainable throughout. In Figures 5.4, 5.5, and 5.6, we show the processing latency of Rhino and Flink before and after a handover takes place (i.e., indicated by the vertical black line). As shown, the latency of query processing (on the left of the black line) is negligibly affected by Rhino's approach in comparison to block replication of Flink and RhinoDFS. As a result, Rhino does not increase processing latency of a query when there is no in-flight reconfiguration.

In Figure 5.7, we report aggregated CPU, memory, network, and disk utilization of our cluster. Rhino and Flink use network to read from Kafka and to perform data exchange and state migration among VMs. They use disk to update state stored in RocksDB and to migrate state among VMs. In contrast, Megaphone requires network for processing and migrations but no disk. Before the reconfiguration, we observe that Rhino and Flink have similar resource utilization during query processing as they execute the same routines. We also notice periodical peaks in all charts, which occur at every checkpoint/replication. In contrast, CPUs usage in Megaphone is constant because of a fiber-based scheduler. Megaphone's memory consumption increases over time since it allocates memory on-demand from the OS.

During a replication, Rhino uses up to 30% network bandwidth and 5% disk write bandwidth more than Flink. However, the higher utilization of network bandwidth results in an up to 3.5x faster state transfer than Flink. Furthermore, Rhino requires 25% less CPU in comparison to Flink, whereas memory consumption is the same. We do not observe spikes in network utilization for Megaphone as it multiplexes state migration with data exchange. After a reconfiguration, Rhino and Flink exhibit similar resource utilization. However, Rhino resumes proactive migration after the reconfiguration (minute 23), while Flink is still resuming query processing. To sum up, the experiment shows that proactive state migration of Rhino does not negatively impact processing latency.

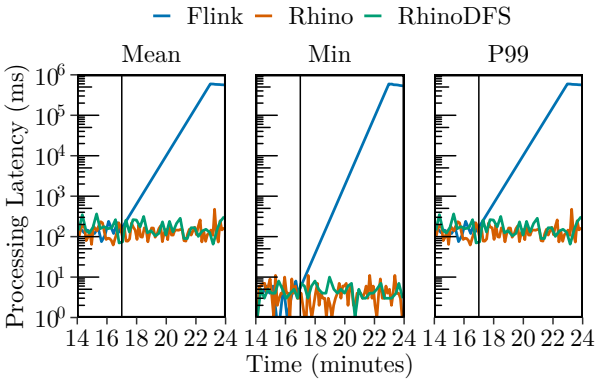
5.4.4 Rhino for Resource Efficiency

In this section, we assess the performance of Rhino, RhinoDFS, and Flink when they perform vertical scaling (see Section 5.4.4.1) and load balancing (see Section 5.4.4.2). We find that horizontal scaling performs similarly to vertical scaling with a slowdown proportional to the size of the state scheduled for migration. Therefore, we omit its evaluation in this section.

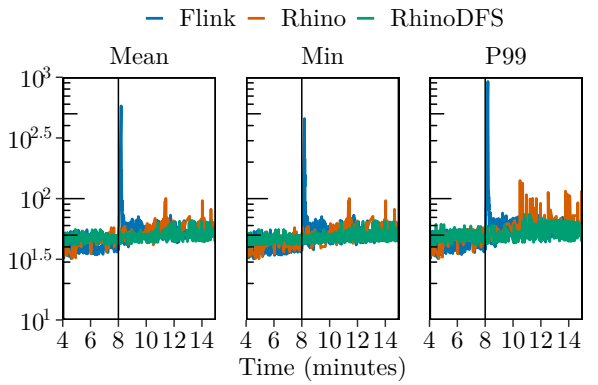
5.4.4.1 Vertical Scaling

In this section, we evaluate the vertical rescaling of NBQ8, NBQ5, and NBQX by adding extra instances on running workers.

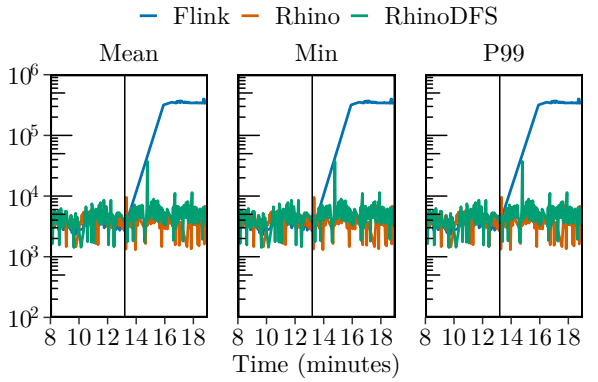
Workload. In this experiment, we run NBQ8, NBQ5, and NBQX on eight VMs (max DOP is 64). We configure each worker to not use full parallelism, i.e., the DOP of the stateful join and aggregation operators is 56 (seven instances per VM). We set the checkpoint interval to two minutes to avoid overhead due to continuous checkpointing. After three checkpoints, we trigger a rescaling operation and switch to full parallelism (64 instances, eight per VM). The overall state



(a) Vertical Scaling NBQ8.



(b) Vertical Scaling NBQ5.



(c) Vertical Scaling NBQX.

Figure 5.5: End-to-end Processing Latency in ms (using log-scale) for our vertical scaling experiments. The vertical black bar represents the moment we trigger a reconfiguration.

size of the last checkpoint before rescaling is approximately 220 GB (NBQ8), 26 MB (NBQ5), and 170 GB (NBQX). After rescaling, we let the SUTs run for three checkpoints and then stop the execution. We collect latency measurements and report them in Figure 5.5a, 5.5b, and 5.5c.

Result. Figures 5.5a, 5.5b, and 5.5c show that the average latency for Rhino's variants is stable. The SUTs can keep average latency of NBQ8 around 130 ms (min: 2 ms, p99: 11 s) for Rhino's variant and 129 ms (min: 2 ms, p99: 9 s) for Flink. In NBQ5, average latency is about 75 ms (min: 2 ms, p99: 119 ms) for all SUTs. For NBQX, average is about 5 s (min: 1 s, p99: 10 s) for all SUTs. As in previous experiment, the high maximum latency is due to the synchronous phase of a checkpoint and higher complexity of NBQX.

Upon rescaling in NBQ8, we observe that latency of Flink increases up to 570 s, whereas RhinoDFS has a sudden spike to 30 s. After scaling, processing latency for Rhino increases to 146 ms in NBQ8. We observe that latency drops to 118 ms after 120 s. In contrast, Flink accumulates ~10 m of latency lag from the data generator in NBQ8. NBQX has similar behavior due to large state sizes. Finally, the execution of NBQ5 is stable in all SUTs before and after the reconfiguration. Upon rescaling, Flink exhibits a spike in latency of 1 s.

Discussion. Overall, this experiment confirms that Rhino supports vertical rescaling without introducing excessive latency on query processing among all queries. In contrast, Flink induces an increased latency of up to three orders of magnitude in NBQ8. This latency spike is significant in NBQ8 as Flink needs to restart a query and reshuffle state among workers. In contrast, Rhino need to checkpoint and migrate ~32 GB of state among workers and have similar performance in NBQ8 and NBQX. When state is small (NBQ5), Flink and Rhino have similar performance because state migration is not a bottleneck.

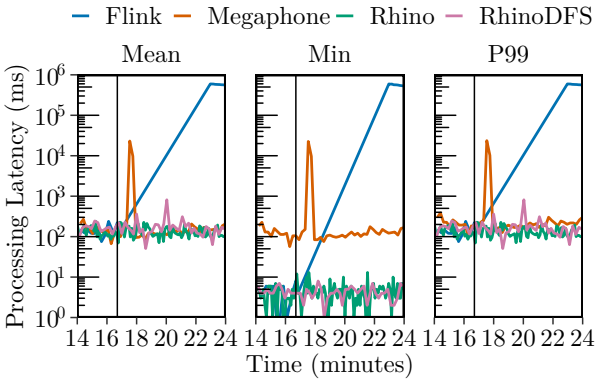
5.4.4.2 Load balancing

In this section, we examine how fast Rhino and Megaphone can reconfigure NBQ8, NBQ5, and NBQX to balance the load among stateful join instances. As there is no implementation of load balancing in Flink, we compare load balancing against vertical scaling.

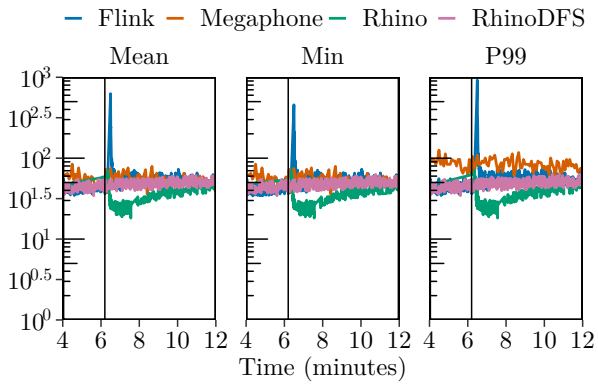
Workload. In this experiment, we deploy NBQ8, NBQ5, and NBQX on eight VMs. We follow the same methodology of the experiment in Section 5.4.4.1. After three checkpoints, we trigger a load balancing operation that moves half virtual nodes from eight instances (one per VM) to other eight instances. We do the same for Megaphone but do not trigger checkpoints.

Result. Our latency measurements in Figures 5.6a, 5.6b, and 5.6c show that Rhino sustains large state stream processing with an average latency of 110 ms in NBQ8. During a load balancing operation, we observe a latency increase of ~60 ms, which Rhino mitigates in one minute. Afterwards, latency is constant with minor fluctuations during checkpointing and proactive state migration. In NBQ5, we observe that Rhino's rebalancing is highly effective such that latency decreases from 60 ms to 20 ms for some minutes. In contrast, the latency of Flink reaches 500 ms for a few seconds after the reconfiguration. In NBQX, we observe that Rhino does not introduce latency after a rebalancing, which stays constant at around 4 sec. Instead, Flink's latency reaches 3.5 min after the reconfiguration. During the reconfiguration of NBQ8 and NBQX, the latency of Megaphone reaches 23.6 s and 10.2 s for ~90 s, respectively.

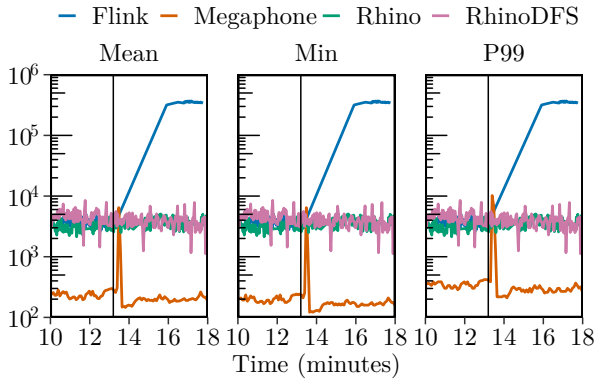
Discussion. This experiment shows that Rhino supports load balancing via migration of virtual nodes with minimal impact on latency in NBQ8, NBQ5, NBQX. This experiments



(a) Load Balancing NBQ8.



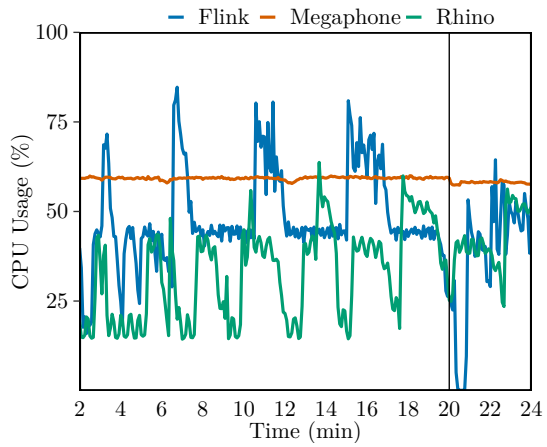
(b) Load Balancing NBQ5.



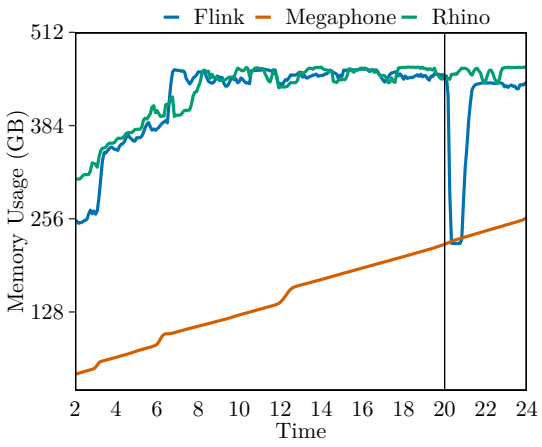
(c) Load Balancing NBQX.

Figure 5.6: End-to-end Processing Latency in ms (using log-scale) for our load balancing experiments. The vertical black bar represents the moment we trigger a reconfiguration.

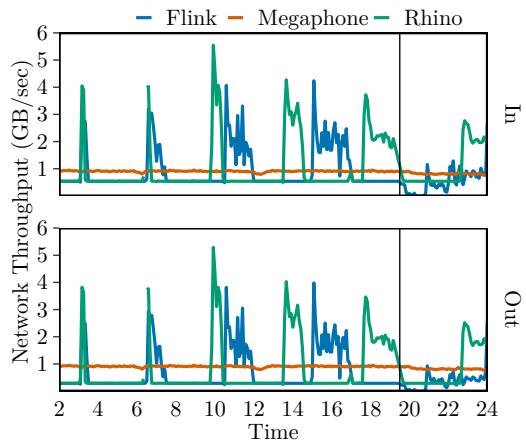
shows that Megaphone's migration affects latency if migrated state is large, i.e., 27 GB. Overall, compared to vertical scaling, our load balancing technique keeps latency constant, whereas Flink shows an increment in latency by three orders of magnitude.



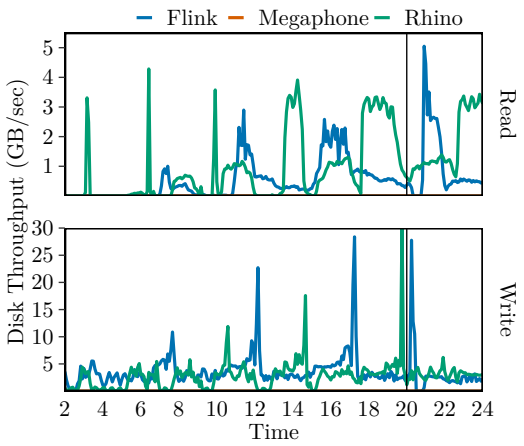
(a) CPU Usage (%).



(b) Memory Usage (GB).



(c) Network Throughput (GB/s).



(d) Disk Throughput (GB/s).

Figure 5.7: Comparison of resource utilization of Apache Flink and Rhino on NBQ8. The black line indicated when we perform a reconfiguration.

5.4.5 Migration under varying data rates

In the following, we show how Rhino supports query processing and state migration under varying data rates. We select NBQ8 for this experiment as it results in larger state.

Workload. We use the same setup as in Section 5.4.2.2 with the following changes. We configure our data generator to produce records at varying speed. Each producer thread initially generates data at 1 MB/s. Every 10 s, data rate increases by 0.5 MB/s until it reaches 8 MB/s (the max sustainable rate for NBQ8). When this happens, data rate decreases by 0.5 MB/s every 10 s, until it reaches 1 MB/s. This repeats throughout the whole experiment. We let Rhino, RhinoDFS, and Flink run until they reach approx. 150 GB of state. Then, we trigger a reconfiguration to migrate 8 operators from one server to the remaining 7 servers.

Result. Our latency measurements in Figure 5.8 show that all SUTs can sustain stream processing under varying data rates. Average latency for all SUTs is 205 ms (min: 9 ms, p99: 826 ms). Upon the reconfiguration, the latency of Rhino and RhinoDFS remains constant, whereas the latency of Flink reaches 225 s. After 2 minutes from the reconfiguration, minimum latency of Flink goes down to 20 ms.

Discussion. Overall, this experiment confirms that Rhino also supports reconfiguration in the presence of fluctuating data rates. We also show that Rhino and Flink are resistant to varying data rates during query processing. However, latency in Rhino is not affected during a reconfiguration, whereas Flink accumulates latency up to 225 s.

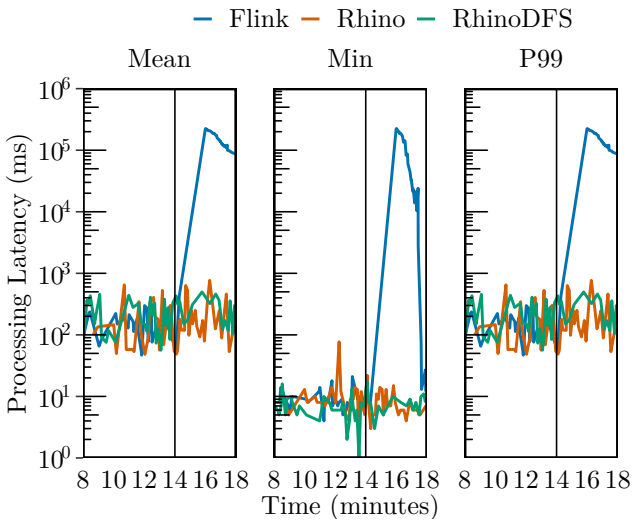


Figure 5.8: End-to-end latency of NBQ8 under varying data rate.

5.4.6 Discussion

The key insights of our evaluation are four-fold. First, we show that Rhino and RhinoDFS can perform a reconfiguration to provide fault-tolerance, vertical scaling, and load balancing with minimal impact on processing latency. When state is large, Rhino achieves up to three orders of magnitude lower latency compared to Flink and two compared to Megaphone. In contrast, if the state size is small, Rhino performs similarly to baseline. Second, we show that Rhino's state migration protocol is beneficial when operator state reaches TB sizes. In particular, Rhino reconfigures a query after a failure 50x faster than Flink, 15x faster than Megaphone, and 11x faster than RhinoDFS. Third, we show that Rhino has minor overhead on OS resources and stream processing. In particular, Rhino uses 30% more network bandwidth than baseline but achieves up to 3.5x faster state transfer. However, we expect our replication runtime to become a bottleneck if an incremental checkpoint to migrate is large, e.g, above 50 GB per instance. We leave investigating this issue as future work, e.g., using adaptive checkpoint scheduling. Fourth, we show that Rhino migrates state and reconfigures a query under fluctuating data rates with no

overhead. Overall, our evaluation shows that Rhino enables on-the-fly reconfigurations of running queries in the presence of large operator state on common workloads.

5.5 Related Work

In the following, we group and describe related work, which we did not fully cover in previous sections.

Fernandez et al. propose SEEP [34] and SDG [34] to address the problem of scaling up and recovering stateful operators in cloud environments. Their experiments target operators with state size up to 200 GB and confirm that larger state leads to higher recovery time. Both approaches are based on migration of checkpointed state, which captures also in-flight records. SEEP stores checkpoints at upstream operators, which may become overwhelmed. SDG improves on SEEP by using incremental, per-operator checkpoints that are persisted to m workers. However, SDG does not control where an operator is resumed, which leads to state transfer and network overhead. In this thesis, we explicitly address these shortcomings to provide low reconfiguration time for queries with very large operator state. To this end, we target large state migration (up to 1 TB) by proactively migrating state to workers where a reconfiguration will take place.

Mai et al. [173] propose Chi to enable user-defined reconfiguration of running queries. Their key idea is the use of control events in the dataflow to trigger a reconfiguration, which reduces latency by 60% on queries with small state. Rhino uses a similar approach, i.e., control events, but it differs from Chi in the following aspects. First, user-defined reconfigurations are orthogonal to Rhino as it supports reconfiguration to transparently provide fault-tolerance, resource elasticity, and load balancing. Second, Chi reactively migrates state in bulk upon a user-defined reconfiguration, which leads to migration time proportional to state size. In contrast, Rhino proactively and incrementally migrates state to provide timely reconfigurations. Finally, a state migration for a logical operator in Chi affects all its physical instances. As a result, all running instances send state to a newly spawned instance. Instead, Rhino migrates state through virtual nodes at a finer granularity to involve the least number of instances.

Megaphone [48] is a system that provides online, fine-grained reconfigurations for SPEs with out-of-band tracking of progress, e.g., Timely Dataflow [21] and MillWheel [14]. There are four key differences that make our contribution different than Megaphone. First, Megaphone and Rhino have different scope. Megaphone enables programmable state migrations in the query DSL, whereas Rhino provides transparent fault-tolerance, resource elasticity, and load balancing via state migration. Second, Megaphone migrates state in bulk and in reaction to user request, whereas Rhino proactively migrates state to reduce reconfiguration time. In particular, Rhino explicitly targets large state migration, while Megaphone handles migration as long as state fits in memory. Third, Megaphone requires out-of-band tracking of progress, i.e., operators must observe each other's progress. Moreover, downstream operators must expose their state to their upstream to perform state migration. While Timely Dataflow provides these features out-of-the-

box, other SPEs need costly synchronization and communication techniques to fullfil Megaphone's requirements. In contrast, Rhino has an internal progress tracking mechanism based on control events that only requires an SPE to follow the streaming dataflow paradigm. Furthermore, Rhino has its own migration protocol, which does not require operators to expose state. As a result, Rhino's protocols could run on Timely Dataflow. Finally, Megaphone and Rhino use different migration protocols. Megaphone uses two migrator operators for every migratable operator. This could lead to scalability issues on larger queries. Instead, Rhino multiplexes state migration of every operator through its distributed runtime.

Clonos [195] achieves fault-tolerant stream processing in the presence of nondeterministic events, while guaranteeing exactly-once processing semantics. Non-deterministic events in an SPE occur when operators require processing-time, timers, random number generators, and checkpoint-based recovery. To this end, Clonos relies on causal logging to store determinants for every nondeterministic event executed by an operator. Upon a failure, Clonos replays events based on the causal log. Rhino has a different scope compared to Clonos. Clonos provides fault-tolerance and high-availability to nondeterministic streaming computations. In contrast, Rhino targets a wider range of use cases including not only fault-tolerance but also resource elasticity and runtime optimizations, such as load balancing. Although nondeterministic events were not covered in this Chapter, extending Rhino to support them is an important research problem that we consider as future work.

Scabbard [196] provides an efficient fault-tolerant mechanism for scale-up stream processing, with limited I/O bandwidth. The contributions of Scabbard's authors are orthogonal to the contributions of this Chapter, because Rhino targets scale-out stream processing and provides a general solution to fault-tolerance, resource elasticity, and runtime optimization, such as load balancing.

Dhalion [180] and DS2 [181] are monitoring tools that trigger scaling decisions. Their contributions are orthogonal to ours as Rhino provides a mechanism to reconfigure a running SPE. We envision an SPE, a monitoring tool, and Rhino that cooperatively handle anomalous operational events to ensure robust stream processing.

5.6 Conclusion

In this Chapter, we have presented Rhino, a system library that enables fine-grained fault-tolerance, resource elasticity, and runtime optimizations in the presence of large distributed state. Through proactive state migration, Rhino removes the bottleneck induced by state transfer upon a reconfiguration. We are currently incorporating Rhino in our new data processing platform NebulaStream [32].

We have evaluated our design choices on a common benchmark suite and compare two variants of Rhino against Flink and Megaphone. Rhino is 50x faster than Flink, 15x faster than Megaphone, and 11x faster than RhinoDFS in reconfiguring a query. Moreover, Rhino shows a reduction in processing latency by up to three orders of magnitude after a reconfiguration. With

Rhino, we enable robust stream processing for queries with very large distributed operator state, regardless of failures or fluctuations in the data rate. Overall, this Chapter lies the foundation for robust stateful stream processing: we envision an efficient SPE (using the contribution of the previous Chapters) that can continuously operate and reconfigure itself to sustain changing workload requirements.

6

Additional Contributions

In this Chapter, we describe further research contributions which have been made while working on this thesis. The following additional contributions are not part of the thesis contents, but are closely related to the topics presented in this thesis:

1. Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl, Volker Markl. On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines. Datenbanksysteme für Business, Technologie und Web (BTW 2019), 2019.

2. Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, Volker Markl. The NebulaStream Platform: Data and Application Management for the Internet of Things. Conference on Innovative Data Systems Research (CIDR), 2020.

3. Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M. Grulich, Ariane Ziehn, Volker Markl. NebulaStream: Complex Analytics Beyond the Cloud. Open Journal of Internet Of Things (OJIOT), 2020.

In *On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines* [197], we introduce techniques to change running stateful streaming queries at runtime. Many major SPEs provide very little functionality to adjust the execution of a potentially infinite streaming query at runtime. Each modification requires a complete query restart, which involves an expensive redistribution of the state of a query and may require external systems to guarantee correct processing semantics. This results in significant downtime, which increase the operational cost of those SPEs. We present a modification protocol that enables modifying specific operators as well as the data flow of a running query while ensuring exactly-once processing semantics. Our modification protocol demonstrates the general feasibility of runtime modifications and paves

the way for many other modification use cases, such as online algorithm tweaking and up-or downscaling operator instances.

In *The NebulaStream Platform: Data and Application Management for the Internet of Things* [31] and *NebulaStream: Complex Analytics Beyond the Cloud* [198], we present NebulaStream: a novel SPE for Internet-of-Things environments. The Internet of Things is a distributed, highly dynamic, and heterogeneous environment of massive scale. Applications for the IoT introduce new challenges for the data management community, as it is necessary to integrate concepts from fog and cloud computing as well as sensor networks in one unified environment. NebulaStream addresses the heterogeneity and distribution of compute and data, supports diverse data and programming models going beyond relational algebra, deals with potentially unreliable communication, and enables constant evolution under continuous operation. In the two papers, we demonstrate the effectiveness of NebulaStream by providing early results on partial aspects.

Furthermore, we mentored a master's thesis title *Rethinking Message Brokers on RDMA and NVM*, which resulted in a publication in the 2020 SIGMOD Student Research Competition [199]. Message brokers orchestrate the exchange of events among different applications, including data-producing services and streaming engines that analyze the data in real-time. Current state-of-the-art message brokers, such as Apache Kafka [200], rely on common Ethernet networks and disk-based storage to receive, save, and send stream records. As a result, they cannot exploit the recent advancements networks and storage technologies, such as RDMA and *Non Volatile Memory* (NVM). Therefore, we explore in this Chapter the application of RDMA and NVM to accelerate message brokers for stream processing workloads. We leveraged RDMA to increase the data transfer throughout between data brokers and SPEs, while we relied on NVM to enable out-of-core storage of stream records in persistent memory. Overall, our approach resulted utilizes up to 80% and 95% of the theoretical network bandwidth of the underlying network hardware for message publishing and consuming, respectively.

7

Conclusion and Future Research

In this thesis, we have identified major performance bottlenecks in the hardware-oblivious design of current SPEs, which do not efficiently execute stateful streaming queries on the modern hardware infrastructures. To meet the needs for low-latency analytics, we have proposed hardware-conscious solutions to revise the architecture of an SPE to enable highly efficient stateful query processing of high-volume, high-speed data streams. To this end, we have presented an analysis of the software architecture as well as the query execution performance of current SPEs to identify open research problems in the stream processing field that prevent them from achieving robust query processing performance. Based on our analysis, we have proposed architectural changes to SPEs to fully leverage the compute and network capabilities as well as the elasticity of modern hardware infrastructures, even in the presence of failures and fluctuating data rates of input streams. Our contributions have at their heart existing research findings from distributed systems, high-performance computing, modern CPUs architecture, and networking domains, which we have extended to apply them to data streaming management.

Overall, this thesis lays the foundation for efficient and reliable stateful stream processing via a hardware-conscious system design that can be applied to SPEs running at cloud scale. We provide building blocks to efficiently execute stateful streaming queries through query compilation and late merge, which achieve high code and data locality on multi-core CPUs equipped with large caches. We propose an architecture re-design of an SPE to co-design stateful stream processing with high-speed RDMA networks to efficiently scale-out query execution. We devise protocols to enable fault-tolerance, runtime optimization, and resource scaling for running stateful queries at runtime via on-the-fly reconfiguration of running stateful streaming queries involving large operator state. Via our solutions, the proposed software prototypes achieve superior performance compared to state-of-the-art SPEs on common streaming workloads. Our solutions can thus be deployed in cloud as well as Internet-of-Things environments to improve the performance of stateful stream processing workloads. In conclusion, the lesson learned from our 5-year research

activities is that a deep understanding of the underlying hardware as well as a rational system design are paramount to provide performance improvements to current and future SPEs (and data management systems at large).

Future Research

The research contributions of this thesis make the case for hardware-conscious acceleration of data stream processing workloads. To offer robust query processing, future SPEs have to efficiently use the more and more powerful hardware resources of the cloud as well as the heterogenous and volatile hardware resources of Internet-of-Things devices. Consequently, the proposed solutions are necessary to efficiently execute stateful queries in a hardware-conscious manner. Our research findings behind scale-up SPEs make the case for query compilation for streaming, which has been investigated by further works [5, 38] after the publication of our research. Our research regarding high-speed networks opens new ways to design and build scale-out SPEs that do not compromise single-node performance to support larger-than-memory workloads. As pointed out by Benson et al. [201], nowadays users have to choose between scale-out and scale-up SPEs - depending on the workload size and requirements. In fact, scale-up SPEs provide hardware-optimized query execution, however, they neglect essential features, such as larger-than-memory state. Using the scale-out execution techniques of Slash as well as our guidelines for scale-up execution, future system builders will be able to accelerate their streaming workloads by fully utilizing the underlying hardware [201]. Furthermore, the state management solutions proposed in this thesis open the way to further research on flexible hardware infrastructures, as industrial setups require to minimize the operational cost of their stream processing workloads. Further works in this area include a control manager for Rhino that allows for adaptive adjustments of running queries based on live statistics from the SPE.

Overall, this thesis paves the way for a new generation of highly-efficient, hardware-conscious SPE for cloud as well as Internet-of-Things environments. Currently, the solutions proposed in this thesis are under integration in the NebulaStream platform for application and data management in the Internet-of-Things [31], which is an on-going research project at the Berlin Institute of Technology (TU Berlin).

References

[1] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777.

[2] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

[3] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 147–156. ISBN: 9781450323765. DOI: 10.1145/2588555.2595641.

[4] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. "StreamBox: Modern Stream Processing on a Multicore Machine". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 617–629. ISBN: 978-1-931971-38-6.

[5] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. "LightSaber: Efficient Window Aggregation on Multi-core Processors". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: ACM, 2020.

[6] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. "BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 705–722. ISBN: 9781450356435. DOI: 10.1145/3299869.3300067.

[7] Tencent. *Oceanus: A one-stop platform for real time stream processing*. 2018. URL: https://www.ververica.com/blog/oceanus-platform-powered-by-apache-flink.

[8] Klaviyo. *Apache Flink Performance Optimization*. 2019. URL: https://klaviyo.tech/flinkperf-c7bd28acc67.

[9] Netflix. *Keystone Real-time Stream Processing Platform*. 2017. URL: https://medium.com/netflix-techblog/keystone-real-time-stream-%5C%5Cprocessing-platform-a3ee651812a.

[10] Uber. *Introducing AthenaX, Uber Engineering's Open Source Streaming Analytics Platform*. 2018. URL: https://eng.uber.com/athenax/.

[11] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, Anirban Banerjee, Benjamin Heintz, Shridar Iyer, and Anshul Jaiswal. "Providing Streaming Joins as a Service at Facebook". In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1809–1821. ISSN: 2150-8097. DOI: 10.14778/3229863.3229869.

[12] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Samza: Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645. ISSN: 2150-8097. DOI: 10.14778/3137765.3137770.

[13] Netflix. *Taming Large State at Netflix*. 2020. URL: https://www.infoq.com/presentations/netflix-event-stream-flink/.

[14] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229.

[15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1792–1803.

[16] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. "Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 659–670. DOI: 10.1109/ICDE.2017.119.

[17] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. "IBM streams processing language: Analyzing big data in motion". In: *IBM Journal of Research and Development* 57.3/4 (2013), pp. 7–1. DOI: 10.1147/JRD.2013.2243535.

[18] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: Easy, Efficient Data-Parallel Pipelines". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 363–375. ISBN: 9781450300193. DOI: 10.1145/1806596.1806638.

[19] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 59–72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[21] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738.

[22] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 147–156. ISBN: 9781450323765. DOI: 10.1145/2588555.2595641.

[23] S. Hauger, T. Wild, A. Mutter, A. Kirstaedter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. "Packet Processing at 100 Gbps and Beyond - Challenges and Perspectives". In: *2009 ITG Symposium on Photonic Networks*. 2009, pp. 1–10.

[24] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940.

[25] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A Catalog of Stream Processing Optimizations". In: *ACM Comput. Surv.* 46.4 (2014), 46:1–46:34. ISSN: 0360-0300. DOI: 10.1145/2528412.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 437–450. ISBN: 978-1-931971-30-0.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. "Datacenter RPCs can be General and Fast". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 1–16. ISBN: 978-1-931971-49-2.

[28] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *Proc. VLDB Endow.* 9.7 (Mar. 2016), pp. 528–539. ISSN: 2150-8097. DOI: 10.14778/2904483.2904485.

[29] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. "On the [Ir]Relevance of Network Performance for Data Processing". In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 126–131.

[30] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777.

[31] Steffen Zeuch, Ankit Chaudhary, Bonaventura Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp Grulich, Sebastian Bress, Jonas Traub, and Volker Markl. "The NebulaStream Platform: Data and Application Management for the Internet of Things". In: *Conference on Innovative Data Systems Research (CIDR)*. 2020.

[32] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Bress, Jonas Traub, and Volker Markl. "The NebulaStream Platform: Data and Application Management for the Internet of Things". In: *CIDR*. 2020.

[33] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. "FlumeJava: Easy, Efficient Data-Parallel Pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010, pp. 363–375.

[34] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 725–736. ISBN: 9781450320375. DOI: 10.1145/2463676.2465282.

[35] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-Tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 423–438. ISBN: 9781450323888. DOI: 10.1145/2517349.2522737.

[37] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Very Large Data Bases*. 2013, pp. 734–746.

[38] Philipp M. Grulich, Bress Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. "Grizzly: Efficient Stream Processing Through Adaptive Query Compilation". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2487–2503. ISBN: 9781450367356. DOI: 10.1145/3318464.3389739.

[39] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. "Trill: A High-performance Incremental Query Processor for Diverse Analytics". In: *Proc. VLDB Endow.* 8.4 (2014), pp. 401–412. ISSN: 2150-8097. DOI: 10.14778/2735496.2735503.

[40] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. "SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 555–569. ISBN: 9781450335317. DOI: 10.1145/2882903.2882906.

[41] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. "Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing". In: *SoCC*. ACM, 2010, pp. 119–130.

[42] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: vol. 51. 1. New York, NY, USA: Association for Computing Machinery, Jan. 2008, pp. 107–113. DOI: 10.1145/1327452.1327492.

[43] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28.

[44] "Apache Storm Trident". In: *URL storm.apache.org/releases/1.1.1/Trident-tutorial.html* ().

[45] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. "The stratosphere platform for big data analytics". In: *The VLDB Journal* 23.6 (2014), pp. 939–964.

[46] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A Catalog of Stream Processing Optimizations". In: *ACM Comput. Surv.* 46.4 (Mar. 2014). ISSN: 0360-0300. DOI: 10.1145/2528412.

[47] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Bress, Tilmann Rabl, and Volker Markl. "Analyzing Efficient Stream Processing on Modern Hardware". In: *Proc. VLDB Endow.* 12.5 (Jan. 2019), pp. 516–530. ISSN: 2150-8097. DOI: 10.14778/3303753.3303758.

[48] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. "Megaphone: Latency-Conscious State Migration for Distributed Streaming Dataflows". In: *Proc. VLDB Endow.* 12.9 (May 2019), pp. 1002–1015. ISSN: 2150-8097. DOI: 10.14778/3329772.3329777.

[49] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 10.

[50] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 743–754. ISBN: 9781450323765. DOI: 10.1145/2588555.2610507.

[51] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. "Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1442–1453. ISSN: 2150-8097. DOI: 10.14778/2824032.2824043.

[52] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 437–450. ISBN: 978-1-931971-30-0.

[53] Mellanox Technologies. *Mellanox ConnectX®6 VPI Card*. https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf. 2019.

[54] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. "FaRM: Fast Remote Memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. ISBN: 978-1-931971-09-6.

[55] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. *Benchmarking Streaming Computation Engines at Yahoo*. Yahoo Engineering Blog: URL https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at. 2015.

[56] *I. T. Association. InfiniBand Roadmap*. Retrieved October 30, 2017, from `http://www.infinibandta.org/`. 2017.

[57] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. "On The [Ir]relevance of Network Performance for Data Processing". In: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, 2016.

[58] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-speed Query Processing over High-speed Networks". In: *Proc. VLDB Endow.* 9.4 (2015), pp. 228–239. ISSN: 2150-8097. DOI: `10.14778/2856318.2856319`.

[59] RDMA Consortium. *RDMA Protocol Verbs Specification (Version 1.0)*. Apr. 2003.

[60] *DiSNI: Direct Storage and Networking Interface*. Retrieved March 30, 2018, from `https://github.com/zrlio/disni`. 2017.

[61] *SparkRDMA ShuffleManager Plugin*. Retrieved March 30, 2018, from `https://github.com/Mellanox/SparkRDMA`. 2017.

[62] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 103–114.

[63] *A SPSC implementation from Facebook*. Retrieved October 30, 2017, from `https://github.com/facebook/folly/tree/master/folly`. 2017.

[64] *A SPSC queue implemented by the Boost library*. Retrieved October 30, 2017, from `www.boost.org/doc/libs/1_64_0/doc/html/boost/lockfree/queue.html`. 2017.

[65] *A SPSC queue implemented by the TBB library*. Retrieved October 30, 2017, from `https://www.threadingbuildingblocks.org/docs/doxygen/a00035.html`. 2017.

[66] *Package java.util.concurrent*. Retrieved October 30, 2017, from `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html`. 2017.

[67] *A ReaderWriter queue which shows superior performance in benchmarks*. Retrieved October 30, 2017, from `https://github.com/cameron314/readerwriterqueue/tree/master/benchmarks`. 2017.

[68] Dmitry Vyukov. "Single-Producer/Single-Consumer Queue". In: *Intel Developer Zonw, URL software.intel.com/en-us/articles/single-producer-single-consumer-queue* (2015).

[69] *Performance optimization for distributed intra-node-parallel streaming systems*. 2013, pp. 62–69. DOI: 10.1109/ICDEW.2013.6547428.

[70] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 36.4 (2015).

[71] Bugra Gedik, Henrique Andrade, and Kun-Lung Wu. "A code generation approach to optimizing high-performance distributed data stream processing". In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*. 2009, pp. 847–856.

[72] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. "Data-Oriented Transaction Execution". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 928–939. ISSN: 2150-8097. DOI: 10.14778/1920841.1920959.

[73] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Making State Explicit for Imperative Big Data Processing". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 49–60. ISBN: 978-1-931971-10-2.

[74] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. "Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 66–77. ISBN: 978-1-4503-5065-5. DOI: 10.1145/3093742.3093925.

[75] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. "General Incremental Sliding-Window Aggregation". In: *Proc. VLDB Endow.* 8.7 (Feb. 2015), pp. 702–713. ISSN: 2150-8097. DOI: 10.14778/2752939.2752940.

[76] Arvind Arasu and Jennifer Widom. "Resource Sharing in Continuous Sliding-Window Aggregates". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 336–347. ISBN: 0120884690.

[77] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch hashing". In: *International Symposium on Distributed Computing*. Springer. 2008, pp. 350–364.

[78] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. "The ART of Practical Synchronization". In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. DaMoN '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450343190. DOI: 10.1145/2933349.2933352.

[79] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. "Continuous Analytics over Discontinuous Streams". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 1081–1092. ISBN: 9781450300322. DOI: 10.1145/1807167.1807290.

[80] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. "Cutty: Aggregate Sharing for User-Defined Windows". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, pp. 1201–1210. ISBN: 9781450340731. DOI: 10.1145/2983323.2983807.

[81] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. "On-the-Fly Sharing for Streamed Aggregation". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 623–634. ISBN: 1595934340. DOI: 10.1145/1142473.1142543.

[82] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams". In: *ACM SIGMOD Record* 34.1 (2005), pp. 39–44.

[83] Jonas Traub, Philipp Grulich, Alejandro Rodríguez Cuéllar, Sebastian Bress, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. "Efficient Window Aggregation with General Stream Slicing". In: *22nd International Conference on Extending Database Technology (EDBT)*. 2019.

[84] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Bress, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. "Scotty: Efficient Window Aggregation for out-of-order Stream Processing". In: *34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1300–1303.

[85] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal* 12.2 (2003), pp. 120–139.

[86] Buğra Gedik. "Generic windowing support for extensible stream processing systems". In: *Software: Practice and Experience* 44.9 (2014), pp. 1105–1128. DOI: https://doi.org/10.1002/spe.2194.

[87] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. "Frames: Data-Driven Windows". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*. DEBS '16. Irvine, California: Association for Computing Machinery, 2016, pp. 13–24. ISBN: 9781450340212. DOI: 10.1145/2933267.2933304.

[88] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. "Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 383–400. ISBN: 978-1-931971-33-1.

[89] Callum Cameron, Jeremy Singer, and David Vengerov. "The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes". In: *Proceedings of the 2015 International Symposium on Memory Management*. ISMM '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 143–156. ISBN: 9781450335898. DOI: 10.1145/2754169.2754180.

[90] Stephan Ewen. *Off-heap Memory in Apache Flink and the curious JIT compiler*. Retrieved November 2, 2018, from https://flink.apache.org/news/2015/09/16/off-heap-memory.html. 2015.

[91] Reynold Xin and Josh Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. Retrieved November 2, 2018, from https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. 2015.

[92] *HotSpot Virtual Machine Garbage Collection Tuning Guide*. Retrieved November 2, 2018, from https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html. 2018.

[93] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. "Benchmarking streaming computation engines: Storm, Flink and Spark streaming". In: *International Parallel and Distributed Processing Symposium, Workshops*. 2016, pp. 1789–1792.

[94] Jamie Grier. *Extending the yahoo! streaming benchmark*. Available at http://data-artisans.com/extending-the-yahoo-streaming-benchmark. 2016.

[95] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. "Linear road: a stream data management benchmark". In: *VLDB*. 2004, pp. 480–491.

[96] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. "Extending XQuery with Window Functions". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 75–86. ISBN: 9781595936493.

[97] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. "Exploiting the Power of Relational Databases for Efficient Stream Processing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: Association for Computing Machinery, 2009, pp. 323–334. ISBN: 9781605584225. DOI: 10.1145/1516360.1516398.

[98] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. "Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL,

USA: Association for Computing Machinery, 2006, pp. 431–442. ISBN: 1595934340. DOI: 10.1145/1142473.1142522.

[99] Peter M. Fischer, Kyumars Sheykh Esmaili, and Renée J. Miller. "Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing". In: *Proceedings of the 13th International Conference on Extending Database Technology*. EDBT '10. Lausanne, Switzerland: Association for Computing Machinery, 2010, pp. 207–218. ISBN: 9781605589459. DOI: 10.1145/1739041.1739068.

[100] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, et al. "Design and Implementation of the MaxStream Federated Stream Processing Architecture". In: (2009).

[101] Erik Zeitler and Tore Risch. "Scalable splitting of massive data streams". In: *DASFAA*. Vol. 5982. 2010, pp. 184–198. ISBN: 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_15.

[102] Erik Zeitler and Tore Risch. "Massive Scale-out of Expensive Continuous Queries". In: *Proc. VLDB Endow.* 4.11 (Aug. 2011), pp. 1181–1188. ISSN: 2150-8097. DOI: 10.14778/3402707.3402752.

[103] Qiming Chen, Meichun Hsu, and Hans Zeller. "Experience in Continuous analytics as a Service (CaaaS)". In: *EDBT*. 2011, pp. 509–514.

[104] Todd Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance*. Available at: http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/. 2015.

[105] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and George Theodorakis. *Do We Need Distributed Stream Processing?* LSDS Blog: https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. 2018.

[106] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 325462-044US. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Aug. 2012.

[107] Michael Armbrust. *Making Apache Spark the Fastest Open Source Streaming Engine*. Databricks Engineering Blog: https://databricks.com/blog/2017/06/06/simple-super-fast-streaming-engine-apache-spark.html. 2017.

[108] Burak Yavuz. *Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems*. Databricks Engineering Blog: https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html. 2017.

[109] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. "The Design of the Borealis Stream Processing Engine." In: *CIDR*. Vol. 5. 2005. 2005, pp. 277–289.

[110] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. "TelegraphCQ: continuous dataflow processing". In: *SIGMOD*. ACM. 2003, pp. 668–668.

[111] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. "NiagaraCQ: A Scalable Continuous Query System for Internet Databases". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: Association for Computing Machinery, 2000, pp. 379–390. ISBN: 1581132174. DOI: 10.1145/342009.335432.

[112] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. "IBM Infosphere Streams for Scalable, Real-Time, Intelligent Transportation Services". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 1093–1104. ISBN: 9781450300322. DOI: 10.1145/1807167.1807291.

[113] Yingjun Wu and Kian-Lee Tan. "ChronoStream: Elastic stateful stream computation in the cloud". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 723–734. DOI: 10.1109/ICDE.2015.7113328.

[114] Peter Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. Vol. 5. 2005, pp. 225–237.

[115] Sebastian Bress, Henning Funke, and Jens Teubner. "Robust Query Processing in Co-Processor-Accelerated Databases". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1891–1906. ISBN: 9781450335317. DOI: 10.1145/2882903.2882936.

[116] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA Database: Data Management for Modern Business Applications". In: *SIGMOD Rec.* 40.4 (Jan. 2012), pp. 45–51. ISSN: 0163-5808. DOI: 10.1145/2094114.2094126.

[117] P.A. Boncz and M. Zukowski. "Vectorwise: Beyond Column Stores". English. In: *IEEE Data Engineering Bulletin* 35.1 (2012), pp. 21–27. ISSN: 1053-1238.

[118] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: vol. 44. 1. New York, NY, USA: Association for Computing Machinery, Jan. 2019. DOI: 10.1145/3283811.

[119] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. "Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 251–264. ISBN: 9781450327589. DOI: 10.1145/2723372.2742783.

[120] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307. ISBN: 978-1-931971-218.

[121] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 2010, pp. 613–624. DOI: 10.1109/ICDE.2010.5447892.

[122] Microsoft Azure. *Microsoft Azure HBv3-series Specification*. https://docs.microsoft.com/en-us/azure/virtual-machines/hbv3-series. 2021.

[123] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. "EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?" In: *Data Management on New Hardware*. DaMoN'22. Philadelphia, PA, USA: Association for Computing Machinery, 2022. ISBN: 9781450393782. DOI: 10.1145/3533737.3538506.

[124] Mellanox Technologies. *Mellanox Quantum® HDR Modular Switch CS8500*. https://www.mellanox.com/files/doc-2020/pb-cs8500.pdf. 2019.

[125] Advanced Micro Devices, Inc. *AMD EPYC® 7003 Series Processor*. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf. 2020.

[126] Feilong Liu, Lingyan Yin, and Spyros Blanas. "Design and Evaluation of an RDMA-Aware Data Shuffling Operator for Parallel Database Systems". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 48–63. ISBN: 9781450349383. DOI: 10.1145/3064176.3064202.

[127] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. "The End of a Myth: Distributed Transactions Can Scale". In: *Proc. VLDB Endow.* 10.6 (Feb. 2017), pp. 685–696. ISSN: 2150-8097. DOI: 10.14778/3055330.3055335.

[128] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 741–758. ISBN: 9781450356435. DOI: 10.1145/3299869.3300081.

[129] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-Value Services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 295–306. ISBN: 9781450328364. DOI: 10.1145/2619239.2626299.

[130] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. "Rack-Scale In-Memory Join Processing Using RDMA". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1463–1475. ISBN: 9781450327589. DOI: 10.1145/2723372.2750547.

[131] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory". In: (2020), pp. 1477–1488. DOI: 10.1109/ICDE48307.2020.00131.

[132] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. "Storm: A Fast Transactional Dataplane for Remote Data Structures". In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR '19. Haifa, Israel: Association for Computing Machinery, 2019, pp. 97–108. ISBN: 9781450367493. DOI: 10.1145/3319647.3325827.

[133] Seokwoo Yang, Siwoon Son, Mi-Jung Choi, and Yang-Sae Moon. "Performance improvement of Apache Storm using InfiniBand RDMA". In: *The Journal of Supercomputing* (2019), pp. 1–27.

[134] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA*. ACM, June 2018.

[135] Barbara Liskov and Rivka Ladin. "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection". In: *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '86. Calgary, Alberta, Canada: Association for Computing Machinery, 1986, pp. 29–39. ISBN: 0897911989. DOI: 10.1145/10590.10593.

[136] Hector Garcia-Molina and Christos A. Polyzios. "Two Epoch Algorithms for Disaster Recovery". In: *Proceedings of the Sixteenth International Conference on Very Large Databases*. Brisbane, Australia: Morgan Kaufmann Publishers Inc., 1990, pp. 222–230.

[137] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456.

[138] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[139] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. "Anna: A KVS for Any Scale". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 401–412. DOI: 10.1109/ICDE.2018.00044.

[140] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams". In: *SIGMOD Rec.* 34.1 (Mar. 2005), pp. 39–44. ISSN: 0163-5808. DOI: 10.1145/1058150.1058158.

[141] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting Coroutines". In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009). ISSN: 0164-0925. DOI: 10.1145/1462166.1462167.

[142] Yongjun He, Jiacheng Lu, and Tianzheng Wang. "CoroBase: Coroutine-Oriented Main-Memory Database Engine". In: *Proc. VLDB Endow.* 14.3 (Nov. 2020), pp. 431–444. ISSN: 2150-8097.

[143] H. T. Kung, Trevor Blackwell, and Alan Chapman. "Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing". In: *Proceedings of the Conference on Communications Architectures, Protocols and Applications*. SIGCOMM '94. London, United Kingdom: Association for Computing Machinery, 1994, pp. 101–114. ISBN: 0897916824. DOI: 10.1145/190314.190324.

[144] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 29–41. ISBN: 9781450309776. DOI: 10.1145/2043556.2043560.

[145] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048.

[146] Justin Levandoski, David Lomet, and Sudipta Sengupta. "LLAMA: A Cache/Storage Subsystem for Modern Hardware". In: *Proceedings of the International Conference on Very Large Databases, VLDB 2013*. VLDB - Very Large Data Bases, Aug. 2013.

[147] Stefan Richter, Victor Alvarez, and Jens Dittrich. "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing". In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 96–107. ISSN: 2150-8097. DOI: 10.14778/2850583.2850585.

[148] Apache Flink. *Apache Flink Configuration*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-master/.

[149] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. "NEXMark - A Benchmark for Queries over Data Streams DRAFT". In: (2004).

[150] John Wilkes. *More Google Cluster Data*. `https://github.com/google/cluster-data`. 2011.

[151] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2471–2486. ISBN: 9781450367356. DOI: `10.1145/3318464.3389723`.

[152] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: `10.1145/42411.42415`.

[153] Ahmad Yasin. "A Top-Down method for performance analysis and counters architecture". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 35–44. DOI: `10.1109/ISPASS.2014.6844459`.

[154] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at what COST?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[155] The Linux Kernel and OpenFabrics Alliance. *Open Fabrics Enterprise Distribution (OFED) Performance Tests*. `https://github.com/linux-rdma/perftest`. 2019.

[156] Ahmad Yasin, Jawad Haj-Yahya, Yosi Ben-Asher, and Avi Mendelson. "A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors". In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019). ISSN: 1544-3566. DOI: `10.1145/3369383`.

[157] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: Association for Computing Machinery, 1995, pp. 1–10. ISBN: 0897917316. DOI: `10.1145/223784.223785`.

[158] *Delivering Application Performance with Oracles InfiniBand Tech*. 2010. URL: `https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf`.

[159] *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*. 2012. URL: `http://www.redbooks.ibm.com/redbooks/pdfs/sg248018.pdf`.

[160] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. "No Compromises: Distributed Transactions with Consistency, Availability, and Performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 54–70. ISBN: 9781450338349.

[161] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. "A Spinning Join That Does Not Get Dizzy". In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. 2010, pp. 283–292. DOI: 10.1109/ICDCS.2010.23.

[162] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. "DFI: The Data Flow Interface for High-Speed Networks". In: *to appear in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021.

[163] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. "High-Performance Design of Hadoop RPC with RDMA over InfiniBand". In: *2013 42nd International Conference on Parallel Processing*. 2013, pp. 641–650. DOI: 10.1109/ICPP.2013.78.

[164] Md. Wasi-ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. DK Panda. "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand". In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 2013, pp. 1908–1917. DOI: 10.1109/IPDPSW.2013.238.

[165] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. "High performance RDMA-based design of HDFS over InfiniBand". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–12. DOI: 10.1109/SC.2012.65.

[166] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. "The Case for RAMCloud". In: *Commun. ACM* 54.7 (July 2011), pp. 121–130. ISSN: 0001-0782. DOI: 10.1145/1965724.1965751.

[167] Tobias Ziegler, Carsten Binnig, and Viktor Leis. "ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 685–699. ISBN: 9781450392495. DOI: 10.1145/3514221.3526187.

[168] Michael Stonebraker, Ugur Cetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *SIGMOD Record* 34 (Dec. 2005), pp. 42–47. DOI: 10.1145/1107499.1107504.

[169] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, Anirban Banerjee, Benjamin Heintz, Shridar Iyer, and Anshul Jaiswal. "Providing Streaming Joins as a Service at Facebook". In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1809–1821. ISSN: 2150-8097. DOI: 10.14778/3229863.3229869.

[170] Ziyu Zhang, Zitan Liu, Qingcai Jiang, Junshi Chen, and Hong An. "RDMA-Based Apache Storm for High-Performance Stream Data Processing". In: *International Journal of Parallel Programming* (2021), pp. 1–14.

[171] Quoc-Cuong To, Juan Soto, and Volker Markl. "A Survey of State Management in Big Data Processing Systems". In: *The VLDB Journal* 27.6 (Dec. 2018), pp. 847–872. ISSN: 1066-8888. DOI: 10.1007/s00778-018-0514-9.

[172] Thomas Heinze, Yuanzhen Ji, Lars Roediger, Valerio Pappalardo, Andreas Meister, Zbigniew Jerzak, and Christof Fetzer. "FUGU: Elastic Data Stream Processing with Latency Constraints". In: *IEEE Data Eng. Bull.* (2015).

[173] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. "Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems". In: *Proc. VLDB Endow.* 11.10 (June 2018), pp. 1303–1316. ISSN: 2150-8097. DOI: 10.14778/3231751.3231765.

[174] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. "Dynamic Plan Migration for Continuous Queries over Data Streams". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 431–442. ISBN: 1581138598. DOI: 10.1145/1007568.1007617.

[175] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, J. Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. "The Stratosphere Platform for Big Data Analytics". In: *The VLDB Journal* (2014).

[176] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43. ISBN: 1581137575. DOI: 10.1145/945445.945450.

[177] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. May 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[178] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. ISBN: 1931971471.

[179] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, and Timothy Roscoe. *Megaphone: Latency-conscious state migration*. https://github.com/strymon-system/megaphone. 2013.

[180] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. "Dhalion: Self-Regulating Stream Processing in Heron". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1825–1836. ISSN: 2150-8097. DOI: 10.14778/3137765.3137786.

[181] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. "Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 783–798. ISBN: 9781931971478.

[182] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922.

[183] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 654–663. ISBN: 0897918886. DOI: 10.1145/258533.258660.

[184] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-Value Store". In: SOSP '07 (2007), pp. 205–220. DOI: 10.1145/1294261.1294281.

[185] Facebook. *RocksDB.org. Facebook Open Source*. 2012. URL: https://rocksdb.org/.

[186] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. "The power of both choices: Practical load balancing for distributed stream processing engines". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 137–148. DOI: 10.1109/ICDE.2015.7113279.

[187] Peter A. Alsberg and John D. Day. "A Principle for Resilient Sharing of Distributed Resources". In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 562–570.

[188] Brian M. Oki and Barbara H. Liskov. "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems". In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '88. Toronto, Ontario, Canada: Association for Computing Machinery, 1988, pp. 8–17. ISBN: 0897912772. DOI: 10.1145/62546.62549.

[189] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 7.

[190] Raghunath Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Andrew Bond, Forrest Carman, and Michael Majdalany. "TPC State of the Council 2013". In: *5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*. 2013.

[191] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. "High-availability algorithms for distributed stream processing". In: *21st International Conference on Data Engineering (ICDE'05)*. 2005, pp. 779–790. DOI: 10.1109/ICDE.2005.72.

[192] Facebook. *RocksDB Tuning Guide*. 2017. URL: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[193] Confluent. *Running Kafka in Production*. 2017. URL: https://docs.confluent.io/current/kafka/deployment.html.

[194] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[195] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. "Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows". In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1637–1650. ISBN: 9781450383431.

[196] Georgios Theodorakis, Fotios Kounelis, Peter Pietzuch, and Holger Pirk. "Scabbard: Single-Node Fault-Tolerant Stream Processing". In: *Proc. VLDB Endow.* 15.2 (Oct. 2021), pp. 361–374. ISSN: 2150-8097. DOI: 10.14778/3489496.3489515.

[197] Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl, and Volker Markl. "On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines". In: LNI P-289 (2019). Ed. by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer, pp. 127–146. DOI: 10.18420/btw2019-09.

[198] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M Grulich, Ariane Ziehn, and Volker Mark. "NebulaStream: Complex analytics beyond the cloud". In: *The International Workshop on Very Large Internet of Things (VLIoT 2020)* (2020).

[199] Hendrik Makait. "Rethinking Message Brokers on RDMA and NVM". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2833–2835. ISBN: 9781450367356. DOI: 10.1145/3318464.3384403.

[200] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. "Building a Replicated Logging System with Apache Kafka". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1654–1655. ISSN: 2150-8097. DOI: 10.14778/2824032.2824063.

[201] Lawrence Benson and Tilmann Rabl. "Darwin: Scale-In Stream Processing". In: *Proceedings of 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA*. 2022.